CHALMERS | GÖTEBORG UNIVERSITY

Technical Report No. 07-15

Proofs Accompanying "Fast and Loose Reasoning is Morally Correct"

NILS ANDERS DANIELSSON

Department of Computer Science and Engineering
Division of Computing Science
CHALMERS UNIVERSITY OF TECHNOLOGY/
GÖTEBORG UNIVERSITY
Göteborg, Sweden, 2007

Abstract

This rough and unpolished document contains detailed proofs supporting the theoretical development in “Fast and Loose Reasoning is Morally Correct”.

Contents

1 Introduction

These notes describe the proofs underlying the theory in "Fast and Loose Reasoning is Morally Correct" (Danielsson et al. 2006, hence-forth referred to as "the paper") in detail.

Note that there are some differences between this text and the paper. In particular, these notes sometimes use different definitions or notation, and the notation is sometimes more sloppy. Furthermore the text is quite rough, with lots of details but little more. There is little point in reading these notes without first consulting the paper.

Note also that the proofs in this text may be overly complicated. In fact, I fully expect that it is possible to simplify (parts of) the development. The aim of this text is not to be beautiful, but to back up the results of the paper.

This document has been compiled by simply including the text from some text files. These files sometimes refer to each other by mentioning a file name ("see definitions"). In order to make this somewhat comprehensible I have listed the names of the files (like definitions) in the table of contents.

The rest of the document is structured as follows:

Section 2 Definitions and proof principles used in the rest of the document.

Section 3 Some properties of functors.

Section 4 Proofs showing that *in* does not need to be defined as a primitive for coinductive types, and similarly that *out* is not needed for inductive types.

Section 5 Proofs relating *in*, *out* and $\sim$.

Section 6 Proof showing that $\sim$ is a PER.

Section 7 It is shown that $\perp$ is not in the domain of the PER (for most types), and also that most types have a non-empty set-theoretic semantic domain.

Section 8 It is shown that *fix* is not in the domain of the PER.

Section 9 The fundamental theorem.

Section 10 The PER is shown to be monotone.

Section 11 It is shown how sizes can be assigned to some values.

Section 12 The approximation lemma is discussed.

Section 13 Explicit characterisations of recursive type formers (μ and ν).

Section 14 It is shown that, when function spaces are not used, two values are related iff they are equal and total. (Recall that, by definition, x is total iff $x \in dom(\sim)$.)

Section 15 Proof that the PER model gives rise to a bicartesian closed category.

Section 16 It is shown that the domain of the PER is empty iff the corresponding set-theoretic semantic domain is empty.

Section 17 The partial surjective homomorphism is defined and shown to be well-defined.

Section 18 Various useful properties that the partial surjective homomorphism satisfies.

Section 19 Proof of the main result.

Section 20 Extension of the main result to a strict language.

2 Definitions

```
Syntax:

  L₁:

    t ::= x | t₁ t₂ | λx.t
        | seq
        | ⋆
        | (,) | fst | snd
        | inl | inr | case
        | in | out | fold | unfold

    σ ::= σ₁ → σ₂ | σ₁ × σ₂ | σ₁ + σ₂ | 1 | μF | νF

    F ::= Id | K_σ | F₁ × F₂ | F₁ + F₂

    Syntactic sugar for terms:

    ∘     ↦ λf g x. f (g x)
    Id    ↦ λf x. f x
    K_σ   ↦ λf x. x
    F + G ↦ λf x. case x (inl ∘ F f) (inr ∘ G f)
    F × G ↦ λf x. seq x (F f (fst x), G f (snd x))

    Syntactic sugar for types:

    Id σ        ↦ σ
    K_τ σ       ↦ τ
    (F + G) σ ↦ F σ + G σ
    (F × G) σ ↦ F σ × G σ

  L₂:

    t ::= ... | fix

Typing rules:

  The obvious ones (see the paper). Note that we allow in to be used
  both for μ-types and ν-types, and similarly for out. However, as
  noted in in-out-proofs we can implement in for ν-types using unfold
  and out, so we do not need to treat that case when doing proofs over
  the syntax of terms. Similar considerations apply to out for
  μ-types.

Semantics:

  ⟦1⟧       = 1_⊥              ⟪1⟫       = 1
  ⟦σ → τ⟧ = ⟨⟦σ⟧ → ⟦τ⟧⟩_⊥  ⟪σ → τ⟫ = ⟪σ⟫ → ⟪τ⟫
  ⟦σ × τ⟧ = (⟦σ⟧ × ⟦τ⟧)_⊥  ⟪σ × τ⟫ = ⟪σ⟫ × ⟪τ⟫
```

$\llbracket \sigma + \tau \rrbracket = (\llbracket \sigma \rrbracket + \llbracket \tau \rrbracket)\_\bot$ $\langle\!\langle \sigma + \tau \rangle\!\rangle = \langle\!\langle \sigma \rangle\!\rangle + \langle\!\langle \tau \rangle\!\rangle$
$\llbracket \mu F \rrbracket$ = The codomain of the initial object in L(F)-Alg(CPO_⊥).
$\langle\!\langle \mu F \rangle\!\rangle$ = The codomain of the initial object in F-Alg(SET).
$\llbracket \nu F \rrbracket$ = The domain of the final object in L(F)-Coalg(CPO).
$\langle\!\langle \nu F \rangle\!\rangle$ = The domain of the final object in F-Coalg(SET).

Here $\langle \cdot \to \cdot \rangle$ is the continuous function space, and $1 = \{\star\}$.

in : F (μ/ν F) → μ/ν F and out : μ/ν F → F (μ/ν F) are defined in CPO, CPO_⊥ and SET, and they are each others inverses. (For CPO and CPO_⊥, replace F by L(F).)

L(F) is inductively defined as follows:

```
L(Id)    = Id
L(K_τ)   = K_τ
L(F × G) = (L(F) × L(G))_⊥
L(F + G) = (L(F) + L(G))_⊥
```

Note that L(F) has the same action on morphisms as $\llbracket F \rrbracket$, where F is the syntactic sugar for terms defined above (see functor-properties).

$\llbracket x \rrbracket\ \Gamma = \Gamma(x)$
$\langle\!\langle x \rangle\!\rangle\ \Gamma = \Gamma(x)$
$\llbracket t_1\ t_2 \rrbracket\ \Gamma = \llbracket t_1 \rrbracket\ \Gamma\ (\llbracket t_2 \rrbracket\ \Gamma)$
$\langle\!\langle t_1\ t_2 \rangle\!\rangle\ \Gamma = \langle\!\langle t_1 \rangle\!\rangle\ \Gamma\ (\langle\!\langle t_2 \rangle\!\rangle\ \Gamma)$
$\llbracket \lambda x.t \rrbracket\ \Gamma = \lambda v.\ \llbracket t \rrbracket\ \Gamma[x \mapsto v]$
$\langle\!\langle \lambda x.t \rangle\!\rangle\ \Gamma = \lambda v.\ \langle\!\langle t \rangle\!\rangle\ \Gamma[x \mapsto v]$
$\llbracket \mathrm{seq} \rrbracket = \lambda v_1\ v_2.\ \{ \bot,\ v_1 = \bot$
$\qquad\qquad\qquad\quad \{ v_2,\ \text{otherwise}$
$\langle\!\langle \mathrm{seq} \rangle\!\rangle = \lambda v_1\ v_2.\ v_2$
$\llbracket \mathrm{fix} \rrbracket = \lambda f.\ \bigsqcup\_(n{\in}\omega)\ f^n\ \bot$
$\langle\!\langle \mathrm{fix} \rangle\!\rangle$ is not defined
$\llbracket \star \rrbracket = \star$
$\langle\!\langle \star \rangle\!\rangle = \star$
$\llbracket (,) \rrbracket = \lambda v_1\ v_2.\ (v_1,\ v_2)$
$\langle\!\langle (,) \rangle\!\rangle = \lambda v_1\ v_2.\ (v_1,\ v_2)$
$\llbracket \mathrm{fst} \rrbracket = \lambda v.\ \{ \bot,\ v = \bot$
$\qquad\qquad\quad \{ v_1,\ v = (v_1,\ v_2)$
$\langle\!\langle \mathrm{fst} \rangle\!\rangle = \lambda (v_1,\ v_2).\ v_1$
$\llbracket \mathrm{snd} \rrbracket = \lambda v.\ \{ \bot,\ v = \bot$
$\qquad\qquad\quad \{ v_2,\ v = (v_1,\ v_2)$
$\langle\!\langle \mathrm{snd} \rangle\!\rangle = \lambda (v_1,\ v_2).\ v_2$
$\llbracket \mathrm{inl} \rrbracket = \lambda v.\ \mathrm{inl}(v)$
$\langle\!\langle \mathrm{inl} \rangle\!\rangle = \lambda v.\ \mathrm{inl}(v)$
$\llbracket \mathrm{inr} \rrbracket = \lambda v.\ \mathrm{inr}(v)$
$\langle\!\langle \mathrm{inr} \rangle\!\rangle = \lambda v.\ \mathrm{inr}(v)$

```
                                 { ⊥,       v = ⊥
⟦case⟧          = λv f_1 f_2.  | f_1 v_1, v = inl(v_1)
                                 { f_2 v_2, v = inr(v_2)
⟨⟨case⟩⟩          = λv f_1 f_2.  { f_1 v_1, v = inl(v_1)
                                 { f_2 v_2, v = inr(v_2)
⟦in⟧            = λv. in(v)
⟨⟨in⟩⟩            = λv. in(v)
⟦out⟧           = λv. out(v)
⟨⟨out⟩⟩           = λv. out(v)
⟦fold_F⟧        = λf. fix (λg. f ∘ L(F) g ∘ out)
⟨⟨fold_F⟩⟩ f      = the unique morphism in F-Alg(SET) from in to f,
                    viewed as a morphism in SET
⟦unfold_F⟧      = λf. fix (λg. in ∘ L(F) g ∘ f)
⟨⟨unfold_F⟩⟩ f    = the unique morphism in F-Coalg(SET) from f to
                    out, viewed as a morphism in SET
```

$$\begin{tikzcd}
\mu F \arrow[r, "\text{fold(f)}"] \arrow[d, shift right, "\text{out}"'] & A \\
F\ \mu F \arrow[r, "\text{F fold(f)}"'] \arrow[u, shift right, "\text{in}"'] & F\ A \arrow[u, "f"']
\end{tikzcd}$$

Note that out : μF → F μF = fold_F (L(F) in), and in : F νF → νF = unfold_F (L(F) out). (For proofs see in-out-proofs.)

Also note that ⟦fold⟧, ⟦unfold⟧, ⟨⟨fold⟩⟩ and ⟨⟨unfold⟩⟩ all satisfy universal properties (see Program Calculation Properties of Continuous Algebras, Fokkinga and Meijer, 1991):

```
  ∀ strict h and f.
    h = ⟦fold⟧ f      ⇔  h ∘ in = f ∘ L(F) h

  ∀ h, f.
    h = ⟦unfold⟧ f    ⇔  out ∘ h = L(F) h ∘ f

  ∀ h, f.
    h = ⟨⟨fold⟩⟩ f      ⇔  h ∘ in = f ∘ F h

  ∀ h, f.
    h = ⟨⟨unfold⟩⟩ f    ⇔  out ∘ h = F h ∘ f
```

PER:

We enforce by definition that ⊥ is not in ~ for function spaces (since ⊥ : μId → μId would be in the domain otherwise).

```
f ~_(σ → τ) g  ⇔  f ≠ ⊥ ≠ g ∧ ∀ x, y ∈ ⟦σ⟧. x ~_σ y ⇒ f x ~_τ g y
x ~_(σ × τ) y  ⇔  ∃ x_1, y_1 ∈ ⟦σ⟧, x_2, y_2 ∈ ⟦τ⟧.
                    x = (x_1, x_2) ∧ y = (y_1, y_2) ∧
                    x_1 ~_σ y_1 ∧ x_2 ~_τ y_2
x ~_(σ + τ) y  ⇔  (∃ x', y' ∈ ⟦σ⟧. x = inl(x') ∧ y = inl(y') ∧ x' ~_σ y') ∨
                  (∃ x', y' ∈ ⟦τ⟧. x = inr(x') ∧ y = inr(y') ∧ x' ~_τ y')
x ~_1 y        ⇔  x = y = ⋆
x ~_μF y       ⇔  (x, y) ∈ μO(F)
x ~_νF y       ⇔  (x, y) ∈ νO(F)
```

O is defined as follows:

```
O(F) : ℘(⟦ μ/ν F ⟧^2) → ℘(⟦ μ/ν F ⟧^2)
O(F)(X) = { (in(a), in(b)) | (a, b) ∈ O'_F(F)(X) }

O'_F(G) : ⟦ μ/ν F ⟧^2 → ℘(⟦ G (μ/ν F) ⟧^2)
O'_F(Id)(X)      = X
O'_F(K_σ)(_)     = { (x, y) | x, y ∈ dom(~_σ), x ~ y }
O'_F(F_1 × F_2)(X) = { ((a_1, b_1), (a_2, b_2)) | (a_1, a_2) ∈ O'_F(F_1)(X),
                                                (b_1, b_2) ∈ O'_F(F_2)(X) }
O'_F(F_1 + F_2)(X) = { (inl(x'), inl(y')) | (x', y') ∈ O'_F(F_1)(X) } ∪
                     { (inr(x'), inr(y')) | (x', y') ∈ O'_F(F_2)(X) }
```

This definition leads to a PER, see per-per.

Note that O(F) is a monotone operator on a complete lattice. We get the following proof principles:

```
Induction:   O(F)(X) ⊆ X  ⇒  μO(F) ⊆ X
Coinduction: X ⊆ O(F)(X)  ⇒  X ⊆ νO(F)
```

How can we use the first principle? Let X be the characteristic set of some property on ℘(⟦ μF ⟧^{2}). If we can show that O(F)(X) ⊆ X, then we know that the property holds for μO(F).

How can we prove that something _is in_ μO(F), then? Since μO(F) is the least prefix point, we would have to prove that it is in all prefix points:

```
  (x_1, x_2) ∈ μO(F)
⇔
  ∀ X ∈ ℘(⟦ μF ⟧^2). O(F)(X) ⊆ X  ⇒  (x_1, x_2) ∈ X
```

Note that we can use the knowledge that μO(F) ⊆ X when proving this:

```
⇔
  ∀ X ∈ ℘(⟦ μF ⟧^2). O(F)(X) ⊆ X ∧ μO(F) ⊆ X  ⇒  (x_1, x_2) ∈ X
```

```
For convenience:

  Define [f] [x] = [f x] (well-defined).

  Using this definition we have extensionality:
    f = g  ⇔  ∀x ∈ [~_σ]. f x = g x
  for arbitrary f, g ∈ [~_(σ → τ)].

Further definitions:

  More definitions can be found in other files. See e.g. biccc,
  partial-surjective-homomorphism and size.
```

3 Some functor properties

The functors satisfy the following properties:

```
 In CPO and CPO_⊥: L(F) = ⟦F⟧. L(F) ⟦σ⟧ = ⟦F σ⟧.

 In SET: F = ⟪F⟫. F ⟪σ⟫ = ⟪F σ⟫.

 In PER: F = [⟦F⟧].
```

(PER is defined in biccc.)

The proofs for SET are simpler variants of the proofs for CPO and CPO_⊥.

- L(F) = ⟦F⟧ (when L(F) is acting on morphisms):

```
  L(Id) f = Id f = f = λx. f x = ⟦Id⟧ f

  L(K_σ) f = K_σ f = λx. x = ⟦K_σ⟧ f

    L(G₁ × G₂) f
  =
    (L(G₁) × L(G₂))_⊥ f
  =
        {⊥, v = ⊥
    λv. │
        {(L(G₁) f x, L(G₂) f y), v = (x, y)
  = { Inductive hypothesis. }
        {⊥, v = ⊥
    λv. │
        {(⟦G₁⟧ f x, ⟦G₂⟧ f y), v = (x, y)
  =
    ⟦G₁ × G₂⟧ f

    L(G₁ + G₂) f
  =
    (L(G₁) + L(G₂))_⊥ f
  =
        {⊥, v = ⊥
    λv. │inl(L(G₁) f x), v = inl(x)
        {inr(L(G₂) f y), v = inr(y)
  = { Inductive hypothesis. }
        {⊥, v = ⊥
    λv. │inl(⟦G₁⟧ f x), v = inl(x)
        {inr(⟦G₂⟧ f y), v = inr(y)
  =
    ⟦G₁ + G₂⟧ f
```

- $L(F)\ \llbracket\sigma\rrbracket = \llbracket F\ \sigma\rrbracket$:

  $L(Id)\ \llbracket\sigma\rrbracket$
  =
  $Id\ \llbracket\sigma\rrbracket$
  =
  $\llbracket\sigma\rrbracket$
  =
  $\llbracket Id\ \sigma\rrbracket$

  $L(K\_\tau)\ \llbracket\sigma\rrbracket$
  =
  $K\_\tau\ \llbracket\sigma\rrbracket$
  =
  $\llbracket\tau\rrbracket$
  =
  $\llbracket K\_\tau\ \sigma\rrbracket$

  $L(G_1 \times G_2)\ \llbracket\sigma\rrbracket$
  =
  $(L(G_1) \times L(G_2))\_\perp\ \llbracket\sigma\rrbracket$
  =
  $((L(G_1) \times L(G_2))\ \llbracket\sigma\rrbracket)\_\perp$
  =
  $(L(G_1)\ \llbracket\sigma\rrbracket \times L(G_2)\ \llbracket\sigma\rrbracket)\_\perp$
  = { Inductive hypothesis. }
  $(\llbracket G_1\ \sigma\rrbracket \times \llbracket G_2\ \sigma\rrbracket)\_\perp$
  =
  $\llbracket G_1\ \sigma \times G_2\ \sigma\rrbracket$

  And analogously for +.

- $F = [\llbracket F\rrbracket]$ (when F is acting on morphisms):

  $[\llbracket Id\rrbracket]\ [f]$
  =
  $[\lambda f\ x.\ f\ x]\ [f]$
  =
  $[\lambda x.\ f\ x]$
  =
  $[f]$
  =
  $Id\ [f]$

  $[\llbracket K\_\sigma\rrbracket]\ [f]$
  =
  $[\lambda f\ x.\ x]\ [f]$
  =
  $[id]$
  =

```
  K_σ [f]

  [⟦G_1 × G_2⟧] [f]
=
  [λf x. ⟦seq⟧ x (⟦G_1⟧ f (⟦fst⟧ x), ⟦G_2⟧ f (⟦snd⟧ x))] [f]
=
  [λx. ⟦seq⟧ x (⟦G_1⟧ f (⟦fst⟧ x), ⟦G_2⟧ f (⟦snd⟧ x))]
= { ∼-equality. }
  [λv. ((⟦G_1⟧ f ∘ ⟦fst⟧) v, (⟦G_2⟧ f ∘ ⟦snd⟧) v)]
= { See biccc for definitions. }
  [⟦G_1⟧ f ∘ ⟦fst⟧] △ [⟦G_2⟧ f ∘ ⟦snd⟧]
= { See biccc for definitions. }
  ([⟦G_1⟧] [f] ∘ fst) △ ([⟦G_2⟧] [f] ∘ snd)
= { Inductive hypothesis. }
  (G_1 [f] ∘ fst) △ (G_2 [f] ∘ snd)
=
  G_1 [f] × G_2 [f]
=
  (G_1 × G_2) [f]

  [⟦G_1 + G_2⟧] [f]
=
  [λf x. ⟦case⟧ x (⟦inl⟧ ∘ ⟦G_1⟧ f) (⟦inr⟧ ∘ ⟦G_2⟧ f)] [f]
=
  [λx. ⟦case⟧ x (⟦inl⟧ ∘ ⟦G_1⟧ f) (⟦inr⟧ ∘ ⟦G_2⟧ f)]
= { See biccc for definitions. }
  [⟦inl⟧ ∘ ⟦G_1⟧ f] ▽ [⟦inr⟧ ∘ ⟦G_2⟧ f]
= { See biccc for definitions. }
  (inl ∘ [⟦G_1⟧] [f]) ▽ (inr ∘ [⟦G_2⟧] [f])
= { Inductive hypothesis. }
  (inl ∘ G_1 [f]) ▽ (inr ∘ G_2 [f])
=
  G_1 [f] + G_2 [f]
=
  (G_1 + G_2) [f]
```

4 Proofs for in and out

```
out : μF → F μF = fold_F (F in)
───────────────────────────────

  out = fold_F (F in)
⇔ { Universality property. }
  out ∘ in = F in ∘ F out
⇔
  out ∘ in = F (in ∘ out)
⇔
  id = F id
⇔
  ⊤

in : F νF → νF = unfold_F (F out)
─────────────────────────────────

  in = unfold_F (F out)
⇔ { Universality property. }
  out ∘ in = F in ∘ F out
⇔ { As above. }
  ⊤
```

```
The category-theoretic proofs above imply the set-theoretic results
  out : μF → F μF = ⟪fold_F (F in)⟫
and
  in : F νF → νF = ⟪unfold_F (F out)⟫,
since F = ⟪F⟫ (see functor-properties), fold = ⟪fold⟫ etc.

Proofs of the same structure can also be used to prove the
domain-theoretic results
  out : μF → F μF = ⟦fold_F (F in)⟧
and
  in : F νF → νF = ⟦unfold_F (F out)⟧,
since L(F) = ⟦F⟧ (see functor-properties), etc., and the functions out
and L(F) in are both strict.

For verbosity we also include explicit proofs for the domain-theoretic
case:

out : μF → F μF = fold_F (L(F) in)
──────────────────────────────────

  fold_F (L(F) in)
=
  fix (λg. L(F) in ∘ L(F) g ∘ out)
=
  fix (λg. L(F) (in ∘ g) ∘ out)
```

```
out is a solution to L(F) (in ∘ g) ∘ out = g. Is it the least
solution? We need to prove that L(F) (in ∘ g) ∘ out = g ⇒ out ⊑ g.

out ⊑ g  ⇔  id ⊑ in ∘ g, and in ∘ g = in ∘ L(F) (in ∘ g) ∘ out

id = fix (λg. in ∘ L(F) g ∘ out), so id ⊑ f for all f satisfying
f = in ∘ L(F) f ∘ out. Done!

in : F νF → νF = unfold_F (L(F) out)
────────────────────────────────────


  unfold_F (L(F) out)
=
  fix (λg. in ∘ L(F) g ∘ L(F) out)
=
  fix (λg. in ∘ L(F) (g ∘ out))

in is a solution to in ∘ L(F) (g ∘ out) = g. Is it the least
solution? We need to prove that in ∘ L(F) (g ∘ out) = g ⇒ in ⊑ g.

in ⊑ g  ⇔  id ⊑ g ∘ out, and g ∘ out = in ∘ L(F) (g ∘ out) ∘ out

Done as above!
```

5 Proofs relating in, out and the PER

```
in x ~_μF in y ⇔      x ~_(F μF) y
   x ~_νF y    ⇔  out x ~_(F νF) out y

The symmetric variants also hold, since in and out are isomorphisms.
```

```
Note first that, by induction over G, if all pairs in X are related,
then all pairs in O'_F(G)(X) are also related.
```

• μF, ⇒:

```
    in x ~ in y : μF
  ⇔ { Def ~, μF fixpoint. }
    (in x, in y) ∈ μO(F) = O(F)(μO(F))
  ⇔ { Def O(F). }
    (x, y) ∈ O'_F(F)(μO(F))
  ⇒ { Initial statement above. }
    x ~ y : F μF
```

□

• μF, ⇐:

```
    ∀ x, y ∈ ⟦F μF⟧. x ~ y ⇒ in x ~ in y
  ⇔
    ∀ x, y ∈ ⟦F μF⟧. x ~ y ⇒ (in x, in y) ∈ μO(F)
  ⇔ { See discussion in definitions. }
    ∀ x, y ∈ ⟦F μF⟧. x ~ y ⇒
      ∀ X ⊆ ⟦μF⟧². μO(F) ⊆ X ∧ O(F)(X) ⊆ X ⇒ (in x, in y) ∈ X
  ⇐ { Transitivity. }
    ∀ x, y ∈ ⟦F μF⟧, X ⊆ ⟦μF⟧².
     x ~ y ∧ μO(F) ⊆ X ⇒ (in x, in y) ∈ O(F)(X)
  ⇔ { Definition of O(F). }
    ∀ x, y ∈ ⟦F μF⟧, X ⊆ ⟦μF⟧².
     x ~ y ∧ μO(F) ⊆ X ⇒ (x, y) ∈ O'_F(F)(X)
  ⇐ { Generalise. }
    ∀ G ≤ F, x, y ∈ ⟦G μF⟧, X ⊆ ⟦μF⟧².
     x ~ y ∧ μO(F) ⊆ X ⇒ (x, y) ∈ O'_F(G)(X)
  ⇐ { Induction over G. }
    ∀ G ≤ F, X ⊆ ⟦μF⟧².
      μO(F) ⊆ X
      ⇒ (∀ G' < G, x, y ∈ ⟦G' μF⟧. x ~ y ⇒ (x, y) ∈ O'_F(G')(X))
      ⇒ ∀ x, y ∈ ⟦G μF⟧. x ~ y ⇒ (x, y) ∈ O'_F(G)(X)
  ⇔ { Case analysis. }
```

  • G = Id:

```
      ∀ X ⊆ ⟦μF⟧², x, y ∈ ⟦μF⟧.
        μO(F) ⊆ X ∧ x ~ y ⇒ (x, y) ∈ X
      ⇔ { (x, y) ∈ μO(F). }
        ⊤
```

- G = K_σ:

```
        ∀ x, y ∈ ⟦σ⟧.
          x ∼ y ⇒ x ∼ y
      ⇔
        ⊤
```

- G = G_1 × G_2:

```
        ∀ X ⊆ ⟦μF⟧^2.
          μO(F) ⊆ X
          ⇒ (∀ G’ < G, x, y ∈ ⟦G’ μF⟧. x ∼ y ⇒ (x, y) ∈ O’_F(G’)(X))
          ⇒ ∀ x_1, y_1 ∈ ⟦G_1 μF⟧, x_2, y_2 ∈ ⟦G_2 μF⟧.
               x_1 ∼ y_1 ∧ x_2 ∼ y_2 ⇒ (x_1, y_1) ∈ O’_F(G_1)(X) ∧ (x_2, y_2) ∈ O’_F(G_2)(X)
      ⇔
        ⊤
```

- G = G_1 + G_2:

```
        ∀ X ⊆ ⟦μF⟧^2.
          μO(F) ⊆ X
          ⇒ (∀ G’ < G, x, y ∈ ⟦G’ μF⟧. x ∼ y ⇒ (x, y) ∈ O’_F(G’)(X))
          ⇒   ∀ x_1, y_1 ∈ ⟦G_1 μF⟧.
                 x_1 ∼ y_1 ⇒ (x_1, y_1) ∈ O’_F(G_1)(X)
             ∧ ∀ x_2, y_2 ∈ ⟦G_2 μF⟧.
                 x_2 ∼ y_2 ⇒ (x_2, y_2) ∈ O’_F(G_2)(X)
      ⇔
        ⊤
```

□

- νF, ⇒:

```
  x ∼ y : νF
⇔ { Def ∼, νF fixpoint. }
  (x, y) ∈ νO(F) = O(F)(νO(F))
⇔ { Def O(F). }
  (out x, out y) ∈ O’_F(F)(νO(F))
⇒ { Initial statement above. }
  out x ∼ out y : F νF
```

□

- νF, ⇐:

```
    ∀ x, y ∈ ⟦νF⟧. out x ∼ out y ⇒ x ∼ y
  ⇔
    ∀ x, y ∈ ⟦νF⟧. out x ∼ out y ⇒ (x, y) ∈ νO(F)
```

```
⇐ { Use coinduction. Let
  {   X = { (x, y) ∈ ⟦νF⟧^2 | out x ~ out y }.
  X ⊆ O(F)(X)
⇔
  ∀ x, y ∈ ⟦νF⟧. out x ~ out y
    ⇒ (x, y) ∈ O(F)(X)
⇔ { in/out are inverses, definition of O(F). }
  ∀ x, y ∈ ⟦F νF⟧. x ~ y ⇒ (x, y) ∈ O'_F(F)(X)
⇐ { Generalise. }
  ∀ G ≤ F, x, y ∈ ⟦G νF⟧. x ~ y ⇒ (x, y) ∈ O'_F(G)(X)
⇐ { Induction over G. }
  ∀ G ≤ F.
    (∀ G' < G, x, y ∈ ⟦G' νF⟧. x ~ y ⇒ (x, y) ∈ O'_F(G')(X))
    ⇒ ∀ x, y ∈ ⟦G νF⟧. x ~ y ⇒ (x, y) ∈ O'_F(G)(X)
⇔ { Case analysis. }
```

- G = Id:

```
    ∀ x, y ∈ ⟦νF⟧. x ~ y ⇒ out x ~ out y
  ⇔ { ⇒ part of proof. }
    ⊤
```

- G = K_σ:

```
    ∀ x, y ∈ ⟦σ⟧. x ~ y ⇒ x ~ y
  ⇔
    ⊤
```

- G = G_1 × G_2:

```
    (∀ G' < G, x, y ∈ ⟦G' νF⟧. x ~ y ⇒ (x, y) ∈ O'_F(G')(X))
    ⇒ ∀ x_1, y_1 ∈ ⟦G_1 νF⟧, x_2, y_2 ∈ ⟦G_2 νF⟧.
        x_1 ~ y_1 ∧ x_2 ~ y_2
        ⇒ (x_1, y_1) ∈ O'_F(G_1)(X) ∧ (x_2, y_2) ∈ O'_F(G_2)(X)
  ⇔
    ⊤
```

- G = G_1 + G_2:

```
    (∀ G' < G, x, y ∈ ⟦G' νF⟧. x ~ y ⇒ (x, y) ∈ O'_F(G')(X))
    ⇒   ∀ x_1, y_1 ∈ ⟦G_1 νF⟧. x_1 ~ y_1 ⇒ (x_1, y_1) ∈ O'_F(G_1)(X)
      ∧ ∀ x_2, y_2 ∈ ⟦G_2 νF⟧. x_2 ~ y_2 ⇒ (x_2, y_2) ∈ O'_F(G_2)(X)
  ⇔
    ⊤
```

□

6 The PER is a PER

Let us now prove that the "PER" is a PER. Since all definitions are symmetric the relation is obviously symmetric. Transitivity follows by induction over the type structure. The only non-trivial cases are those for μF and νF.

```
μF
──

∀ F, x, y, z ∈ ⟦μF⟧.
  x ~ y : μF ∧ y ~ z : μF  ⇒  x ~ z : μF

⇔ Definition ~.

∀ F, x, y, z ∈ ⟦μF⟧.
  (x, y) ∈ μO(F) ∧ y ~ z : μF  ⇒  x ~ z : μF

⇐ Proof by induction. Define X_F ≡
    { (x, y) | x, y ∈ ⟦μF⟧ ∧ x ~ y ∧ (∀ z ∈ ⟦μF⟧. y ~ z ⇒ x ~ z) }.
  Prove that O(F)(X_F) ⊆ X_F which implies that μO(F) ⊆ X_F.

∀ F, x, y ∈ ⟦μF⟧.
  (x, y) ∈ O(F)(X_F) ⇒ x ~ y ∧ (∀ z ∈ ⟦μF⟧. y ~ z ⇒ x ~ z)

⇔ Rewrite, in/out bijections, see also per-and-in-out.

∀ F, x, y, z ∈ ⟦F μF⟧.
  (x, y) ∈ O'_F(F)(X_F) ⇒ x ~ y ∧ y ~ z ⇒ x ~ z

⇐ Generalise.

∀ F, G ≤ F, x, y, z ∈ ⟦G μF⟧.
  (x, y) ∈ O'_F(G)(X_F) ⇒ x ~ y ∧ y ~ z ⇒ x ~ z

⇐ Induction over G.

∀ F, G ≤ F.
  (∀ G' < G, x, y, z ∈ ⟦G' μF⟧.
    (x, y) ∈ O'_F(G')(X_F) ⇒ x ~ y ∧ y ~ z ⇒ x ~ z)
  ⇒ ∀ x, y, z ∈ ⟦G μF⟧.
    (x, y) ∈ O'_F(G)(X_F) ⇒ x ~ y ∧ y ~ z ⇒ x ~ z

⇐ Case on G.

• G = Id:

  ⇐ Nothing < Id, definition O'_F(Id) and Id.
```

```
∀ F, x, y, z ∈ ⟦μF⟧.
  x ~ y ∧ y ~ z ⇒ x ~ z
  ⇒ x ~ y ∧ y ~ z ⇒ x ~ z

⇐ Assumption.

⊤
```

- G = K_σ ≤ F:

```
⇐ Nothing < K_σ, definition O'_F(K_σ) and K_σ.

∀ x, y, z ∈ ⟦σ⟧.
  x ~ y ⇒ x ~ y ∧ y ~ z ⇒ x ~ z

⇐ Assumption.

∀ x, y, z ∈ ⟦σ⟧.
  x ~ y ∧ y ~ z ⇒ x ~ z

⇐ Outer inductive hypothesis (σ < νF).

⊤
```

- G = G_1 × G_2:

```
⇐ Definition O'_F($G_1$ × $G_2$).

∀ F.
  (∀ G' < G, x, y, z ∈ ⟦G' μF⟧.
    (x, y) ∈ O'_F(G')(X_F) ⇒ x ~ y ∧ y ~ z ⇒ x ~ z)
  ⇒ ∀ x, y, z ∈ ⟦G μF⟧.
    (x, y) ∈ { (($a_1$, $b_1$), ($a_2$, $b_2$)) | ($a_1$, $a_2$) ∈ O'_F($G_1$)(X_F),
                                             ($b_1$, $b_2$) ∈ O'_F($G_2$)(X_F) }
    ⇒ x ~ y ∧ y ~ z ⇒ x ~ z

⇔ Rewrite. z ∈ ⟦($G_1$ × $G_2$) μF⟧ and ($a_2$, $b_2$) ~ z implies that
  z = ($z_1$, $z_2$).

∀ F.
  (∀ G' < G, x, y, z ∈ ⟦G' μF⟧.
    (x, y) ∈ O'_F(G')(X_F) ⇒ x ~ y ∧ y ~ z ⇒ x ~ z)
  ⇒ ∀ $a_1$, $a_2$, $z_1$ ∈ ⟦$G_1$ μF⟧, $b_1$, $b_2$, $z_2$ ∈ ⟦$G_2$ μF⟧.
    ($a_1$, $a_2$) ∈ O'_F($G_1$)(X_F) ∧ ($b_1$, $b_2$) ∈ O'_F($G_2$)(X_F)
    ⇒ ($a_1$, $b_1$) ~ ($a_2$, $b_2$) ∧ ($a_2$, $b_2$) ~ ($z_1$, $z_2$) ⇒ ($a_1$, $b_1$) ~ ($z_1$, $z_2$)

⇐ Specialise.
```

```
∀ F.
  ((∀ $a_1$, $a_2$, $z_1$ ∈ ⟦$G_1$ μF⟧.
     ($a_1$, $a_2$) ∈ O'_F($G_1$)(X_F) ⇒ $a_1$ ∼ $a_2$ ∧ $a_2$ ∼ $z_1$ ⇒ $a_1$ ∼ $z_1$)
   ∧
   (∀ $b_1$, $b_2$, $z_2$ ∈ ⟦$G_2$ μF⟧.
     ($b_1$, $b_2$) ∈ O'_F($G_2$)(X_F) ⇒ $b_1$ ∼ $b_2$ ∧ $b_2$ ∼ $z_2$ ⇒ $b_1$ ∼ $z_2$)
   )
  )
  ⇒ ∀ $a_1$, $a_2$, $z_1$ ∈ ⟦$G_1$ μF⟧, $b_1$, $b_2$, $z_2$ ∈ ⟦$G_2$ μF⟧.
    ($a_1$, $a_2$) ∈ O'_F($G_1$)(X_F) ∧ ($b_1$, $b_2$) ∈ O'_F($G_2$)(X_F)
    ⇒ ($a_1$, $b_1$) ∼ ($a_2$, $b_2$) ∧ ($a_2$, $b_2$) ∼ ($z_1$, $z_2$) ⇒ ($a_1$, $b_1$) ∼ ($z_1$, $z_2$)

⇐ Assumption, definition of ∼.

⊤
```

- G = G_1 + G_2:

```
⇐ Definition O'_F($G_1$ + $G_2$).

∀ F.
  (∀ G' < G, x, y, z ∈ ⟦G' μF⟧.
    (x, y) ∈ O'_F(G')(X_F) ⇒ x ∼ y ∧ y ∼ z ⇒ x ∼ z)
  ⇒ (∀ x, y ∈ ⟦$G_1$ μF⟧, z ∈ ⟦G μF⟧.
      (x, y) ∈ O'_F($G_1$)(X_F) ⇒ inl(x) ∼ inl(y) ∧ inl(y) ∼ z ⇒ inl(x) ∼ z)
    ∧
    (∀ x, y ∈ ⟦$G_2$ μF⟧, z ∈ ⟦G μF⟧.
      (x, y) ∈ O'_F($G_2$)(X_F) ⇒ inr(x) ∼ inr(y) ∧ inr(y) ∼ z ⇒ inr(x) ∼ z)

⇔ Definition ∼, inl(y) ∼ z implies that z = inl(z'), and similarly
  for inr.

∀ F.
  (∀ G' < G, x, y, z ∈ ⟦G' μF⟧.
    (x, y) ∈ O'_F(G')(X_F) ⇒ x ∼ y ∧ y ∼ z ⇒ x ∼ z)
  ⇒ (∀ x, y, z ∈ ⟦$G_1$ μF⟧.
      (x, y) ∈ O'_F($G_1$)(X_F) ⇒ x ∼ y ∧ y ∼ z ⇒ x ∼ z)
    ∧
    (∀ x, y, z ∈ ⟦$G_2$ μF⟧.
      (x, y) ∈ O'_F($G_2$)(X_F) ⇒ x ∼ y ∧ y ∼ z ⇒ x ∼ z)

⇐ Assumption.

⊤
```

νF
—

```
∀ F, x, y, z ∈ ⟦νF⟧.
  x ∼ y ∧ y ∼ z  ⇒  x ∼ z
```

```
⇔ Definition ∼.

∀ F, x, y, z ∈ ⟦νF⟧.
  x ∼ y ∧ y ∼ z  ⇒  (x, z) ∈ νO(F)

⇔ Rewrite.

∀ F, x, z ∈ ⟦νF⟧.
  (∃ y ∈ ⟦νF⟧. x ∼ y ∧ y ∼ z)  ⇒  (x, z) ∈ νO(F)

⇐ Proof by coinduction. Define X_F ≡
    { (x, z) ∈ ⟦νF⟧² | ∃ y ∈ ⟦νF⟧. x ∼ y ∧ y ∼ z }.
  Prove that X_F ⊆ O(F)(X_F) which implies that X_F ⊆ νO(F).

∀ F, x, z ∈ ⟦νF⟧.
  (x, z) ∈ X_F ⇒ (x, z) ∈ O(F)(X_F)

⇔ Rewrite.

∀ F, x, y, z ∈ ⟦νF⟧.
  x ∼ y ∧ y ∼ z ⇒ (x, z) ∈ O(F)(X_F)

⇔ in/out bijections, see also per-and-in-out.

∀ F, x, y, z ∈ ⟦F νF⟧.
  x ∼ y ∧ y ∼ z ⇒ (x, z) ∈ O'_F(F)(X_F)

⇐ Generalise.

∀ F, G ≤ F, x, z ∈ ⟦G νF⟧.
  x ∼ y ∧ y ∼ z ⇒ (x, z) ∈ O'_F(G)(X_F)

⇐ Induction over G.

∀ F, G ≤ F.
  (∀ G' < G, x, y, z ∈ ⟦G' νF⟧.
    x ∼ y ∧ y ∼ z ⇒ (x, z) ∈ O'_F(G')(X_F))
  ⇒ ∀ x, y, z ∈ ⟦G νF⟧.
    x ∼ y ∧ y ∼ z ⇒ (x, z) ∈ O'_F(G)(X_F)

⇐ Case on G.
```

- G = Id:

```
  ⇐ Nothing < Id, definition O'_F(Id) and Id.

  ∀ F, x, y, z ∈ ⟦νF⟧.
    x ∼ y ∧ y ∼ z ⇒ (x, z) ∈ X_F
```

```
⇔ Definition X_F, assumption.

⊤
```

- `G = K_σ ≤ F:`

```
⇐ Nothing < K_σ, definition O'_F(K_σ) and K_σ.

∀ x, y, z ∈ ⟦σ⟧.
  x ∼ y ∧ y ∼ z ⇒ x ∼ z

⇐ Outer inductive hypothesis (σ < νF).

⊤
```

- `G = G_1 × G_2:`

```
⇐ Definition O'_F(G_1 × G_2).

∀ F.
  (∀ G' < G, x, y, z ∈ ⟦G' νF⟧.
    x ∼ y ∧ y ∼ z ⇒ (x, z) ∈ O'_F(G')(X_F))
  ⇒ ∀ x, y, z ∈ ⟦G νF⟧.
    x ∼ y ∧ y ∼ z
    ⇒ (x, z) ∈ { ((a_1, b_1), (a_2, b_2)) | (a_1, a_2) ∈ O'_F(G_1)(X_F),
                                            (b_1, b_2) ∈ O'_F(G_2)(X_F) }

⇔ x, y, z ∈ ⟦(G_1 × G_2) νF⟧ ∩ dom(∼) implies that x = (a_1, b_1),
  y = (a_2, b_2), z = (a_3, b_3). Rewrite.

∀ F.
  (∀ G' < G, x, y, z ∈ ⟦G' νF⟧.
    x ∼ y ∧ y ∼ z ⇒ (x, z) ∈ O'_F(G')(X_F))
  ⇒ ∀ a_1, a_2, a_3 ∈ ⟦G_1 νF⟧, b_1, b_2, b_3 ∈ ⟦G_2 νF⟧.
    (a_1, b_1) ∼ (a_2, b_2) ∧ (a_2, b_2) ∼ (a_3, b_3)
    ⇒ (a_1, a_3) ∈ O'_F(G_1)(X_F) ∧ (b_1, b_3) ∈ O'_F(G_2)(X_F)

⇔ Definition ∼.

∀ F.
  (∀ G' < G, x, y, z ∈ ⟦G' νF⟧.
    x ∼ y ∧ y ∼ z ⇒ (x, z) ∈ O'_F(G')(X_F))
  ⇒ ∀ a_1, a_2, a_3 ∈ ⟦G_1 νF⟧, b_1, b_2, b_3 ∈ ⟦G_2 νF⟧.
    a_1 ∼ a_2 ∧ a_2 ∼ a_3 ∧ b_1 ∼ b_2 ∧ b_2 ∼ b_3
    ⇒ (a_1, a_3) ∈ O'_F(G_1)(X_F) ∧ (b_1, b_3) ∈ O'_F(G_2)(X_F)

⇐ Assumption.

⊤
```

- $G = G_1 + G_2$:

```
⇐ Definition O'_F(G₁ + G₂).

∀ F.
  (∀ G' < G, x, y, z ∈ ⟦G' νF⟧.
    x ~ y ∧ y ~ z ⇒ (x, z) ∈ O'_F(G')(X_F))
  ⇒ ∀ x, y, z ∈ ⟦G νF⟧.
    x ~ y ∧ y ~ z
    ⇒ (x, z) ∈   { (inl(x'), inl(z') | (x', z') ∈ O'_F(G₁)(X_F) }
                ∪ { (inr(x'), inr(z') | (x', z') ∈ O'_F(G₂)(X_F) }

⇐ x ~ y : (G₁ + G₂) νF implies that x = inl(x₁), y = inl(y₁) or
  x = inr(x₂), y = inr(y₂), and similarly for y and z. Definition
  of ~.

∀ F.
  (∀ G' < G, x, y, z ∈ ⟦G' νF⟧.
    x ~ y ∧ y ~ z ⇒ (x, z) ∈ O'_F(G')(X_F))
  ⇒ ((∀ x₁, y₁, z₁ ∈ ⟦G₁ νF⟧.
        x₁ ~ y₁ ∧ y₁ ~ z₁ ⇒ (x₁, z₁) ∈ O'_F(G₁)(X_F))
     ∧
     (∀ x₂, y₂, z₂ ∈ ⟦G₂ νF⟧.
        x₂ ~ y₂ ∧ y₂ ~ z₂ ⇒ (x₂, z₂) ∈ O'_F(G₂)(X_F))
    )

⇐ Assumption.

⊤
```

7 Some types are troublesome

Theorem: ⊥ ∈ dom(~_σ) iff σ is generated by the following grammar:

```
  χ ::= νId | μK_χ | νK_χ
```

Corollary: If νId isn't used, then ⊥ ∉ dom(~).

Note that ⟦χ⟧ = {⊥} for all these types.

Proof:

⇒: By induction over the structure of σ.

For all types except μF, νF we have ⊥ ∉ dom(~σ) by definition.

For μ/ν we have the following:

```
   ⊥ ∈ dom(~)

  ⇒

   (⊥, ⊥) ∈ μ/νO(F)

  ⇒ { Fixpoint, μ/νO(F) = O(F)(μ/νO(F)). }

   (⊥, ⊥) ∈ O(F)(μ/νO(F))

  ⇒ { out is strict. }

   (⊥, ⊥) ∈ O'_F(F)(μ/νO(F))
```

Now let us proceed by case analysis on F:

- F = Id:

  νId is generated by the grammar.

  μId is not, but ⊥ ∉ dom(~_μId) since the empty set is a fixpoint of O(Id), so we get a contradiction.

- F = K_τ:

  We have O'_F(F)(μ/νO(F)) = { (x, y) | x, y ∈ dom(~_τ), x ~ y }, so (⊥, ⊥) ∈ O'_F(F)(μ/νO(F)) implies that ⊥ ∈ dom(~_τ). By the inductive hypothesis we get that τ is generated by the grammar, and this implies that μ/ν K_τ is.

- F = $F_1 \times F_2$ or $F_1 + F_2$:

  By the definition of O'_F(F) we immediately get that $(\perp, \perp) \notin$ O'_F(F)(μ/νO(F)), so we have a contradiction.

$\Leftarrow$: By induction over the structure of the grammar.

- χ = νId:

  We need to show that $(\perp, \perp) \in$ νO(Id), and by coinduction this is true if X $\subseteq$ O(Id)(X) for some X $\in \wp(⟦$ νId $⟧^2)$ with $(\perp, \perp) \in$ X. This is satisfied by X = $\{(\perp, \perp)\}$.

- χ = μK_χ:

  By inductive hypothesis we get that $\perp \in$ dom(~_χ). This implies that $(\perp, \perp) \in$ O'_K_χ(K_χ)(X) for any X, in particular for μO(K_χ). Hence, by strictness of in, $(\perp, \perp) \in$ O(K_χ)(μK_χ) = μK_χ.

- χ = νK_χ:

  Analogously to the μK_χ case.

Theorem: ⟪σ⟫ $\neq \emptyset$ for types defined according to the following grammar:

σ ::= 1 | $\sigma_1 \times \sigma_2$ | $\sigma_1 + \sigma_2$ | $\sigma_1 \to \sigma_2$ | μF' | νF'

F' ::= K_σ | F' $\times$ F | F $\times$ F' | F' + F | F + F'

F ::= Id | K_σ | F $\times$ F | F + F

Note that a type belongs to this grammar iff it syntactically contains 1 (proof by induction).

Proof: By induction over the type structure.

Easy for 1, $\times$, +, $\to$.

For μF':

Recall:

⟪μF⟫ = The codomain of the initial object in F-Alg(SET).

Since the total functions in and out both exist, we know that ⟪μF'⟫ = $\emptyset \Leftrightarrow$ ⟪F' μF'⟫ = $\emptyset$. We will now prove that ⟪μF'⟫ = $\emptyset \Leftrightarrow$ ⟪F'' μF'⟫ = $\emptyset$ is impossible for functors F'' of the restricted kind defined above.

Proof by induction over structure of F'':

F'' = Id: Impossible.

F'' = K_τ (with $\langle\langle\tau\rangle\rangle = \emptyset$): Impossible (by inductive hypothesis).

F'' = K_τ (with $\langle\langle\tau\rangle\rangle \neq \emptyset$): Done.

F'' = $F_1 \times F_2$ or $F_1 + F_2$ (with at least one of F_1 and F_2 restricted as above): Done by inductive hypothesis since $\langle\langle$F'' μF'$\rangle\rangle = \emptyset$ iff $\langle\langle F_1$ μF'$\rangle\rangle = \emptyset$ and $\langle\langle F_2$ μF'$\rangle\rangle = \emptyset$.

For νF: Similarly.

8 The function `fix` is not in the PER

```
For most types σ. fix ∉ dom(~_σ)
────────────────────────────────


Proof:

id ∈ dom(~), so fix ∈ dom(~) would imply that fix id = ⊥ ∈ dom(~),
which it does not for most types (see troublesome-types).
```

9 The fundamental theorem

If t does not contain uses of seq at type $\sigma \to \tau \to \tau$, where dom(~_σ) includes $\bot$, then the following is true:

$(\forall\ x \in FV(t).\ \Gamma_1(x) \sim \Gamma_2(x))$
$\Rightarrow [\![\ t\ ]\!]\ \Gamma_1 \sim [\![\ t\ ]\!]\ \Gamma_2.$

Proof by induction over structure of t.
Inductive hypothesis:
$\forall\ t' < t,\ \Gamma_1',\ \Gamma_2'.$
t' $\neq$ seq at the wrong types $\wedge$
$(\forall\ x \in FV(t').\ \Gamma_1'(x) \sim \Gamma_2'(x))$
$\Rightarrow [\![\ t'\ ]\!]\ \Gamma_1' \sim [\![\ t'\ ]\!]\ \Gamma_2'.$

- t = x: By assumption.

- $t = t_1\ t_2$:

  $[\![\ t_1\ t_2\ ]\!]\ \Gamma_1$
  =
  $([\![\ t_1\ ]\!]\ \Gamma_1)\ ([\![\ t_2\ ]\!]\ \Gamma_1)$
  $\sim$ Inductive hypothesis twice, definition of $\sim$.
  $([\![\ t_1\ ]\!]\ \Gamma_2)\ ([\![\ t_2\ ]\!]\ \Gamma_2)$
  =
  $[\![\ t_1\ t_2\ ]\!]\ \Gamma_2$

- $t = \lambda x.\ t_1$:

  $[\![\ \lambda x.\ t_1\ ]\!]\ \Gamma_1$
  =
  $\lambda v.\ [\![\ t_1\ ]\!]\ \Gamma_1[x \mapsto v]$
  $\sim$ Inductive hypothesis, definition of $\sim$.
  $\lambda v.\ [\![\ t_1\ ]\!]\ \Gamma_2[x \mapsto v]$
  =
  $[\![\ \lambda x.\ t_1\ ]\!]\ \Gamma_2$

For the rest we don't have to use the inductive hypothesis.

- $[\![\ seq\ ]\!] \sim [\![\ seq\ ]\!]$
  $\Leftrightarrow$
  $\forall\ x_1 \sim x_2 : \sigma,\ y_1 \sim y_2 : \sigma'.\ [\![\ seq\ ]\!]\ x_1\ y_1 \sim [\![\ seq\ ]\!]\ x_2\ y_2$
  $\Leftrightarrow \bot \notin$ dom(~_σ) by assumption.
  $\top$

- $[\![\ 1\ ]\!] \sim [\![\ 1\ ]\!]$
  $\Leftrightarrow$ By definition.
  $\top$

- $[\![$ (,) $]\!] \sim [\![$ (,) $]\!]$
  $\Leftrightarrow$
  $\forall\ x_1 \sim x_2,\ y_1 \sim y_2.\ (x_1,\ x_2) \sim (y_1,\ y_2)$
  $\Leftrightarrow$
  $\top$

- $[\![$ fst $]\!] \sim [\![$ fst $]\!]$
  $\Leftrightarrow$
  $\forall\ p \sim q.\ [\![$ fst $]\!]\ p \sim [\![$ fst $]\!]\ q$
  $\Leftrightarrow$ $p \sim q : \sigma \times \tau \Rightarrow p \neq \bot \neq q.$
  $\top$

- snd analogous.

- $[\![$ inl $]\!] \sim [\![$ inl $]\!]$
  $\Leftrightarrow$
  $\forall\ x \sim y.$ inl(x) $\sim$ inl(y)
  $\Leftrightarrow$
  $\top$

- inr analogous.

- $[\![$ case $]\!] \sim [\![$ case $]\!]$
  $\Leftrightarrow$
  $\forall\ x_1 \sim x_2,\ f_1 \sim f_2,\ g_1 \sim g_2.$
  $[\![$ case $]\!]\ x_1\ f_1\ g_1 \sim [\![$ case $]\!]\ x_2\ f_2\ g_2$
  $\Leftrightarrow$ $x_1 \sim x_2 : \sigma + \tau \Rightarrow x_1 \neq \bot \neq x_2.$
  $\top$

- $[\![$ in $]\!] \sim [\![$ in $]\!]$
  $\Leftrightarrow$
  $\forall\ x \sim y.\ [\![$ in $]\!]\ x \sim [\![$ in $]\!]\ y$
  $\Leftrightarrow$ See per-and-in-out.
  $\forall\ x \sim y.\ x \sim y$
  $\Leftrightarrow$
  $\top$

- $[\![$ out $]\!] \sim [\![$ out $]\!]$
  $\Leftrightarrow$
  $\forall\ x \sim y.\ [\![$ out $]\!]\ x \sim [\![$ out $]\!]\ y$
  $\Leftrightarrow$ See per-and-in-out.
  $\forall\ x \sim y.\ x \sim y$
  $\Leftrightarrow$
  $\top$

- fold:

  F: Polynomial functor on CPO_⊥.
  fold_F = λf. fix (λg. f ∘ L(F) g ∘ out)

```
∀ F. fold ~ fold

⇔ Def ~.

∀ F, f ~ g : F τ → τ, x ~ y : μF. fold f x ~ fold g y

⇐ Given F, f, g, let X ⊆ ℘(⟦ μF ⟧²) be the set of all pairs
   (x, y) with x, y : μF for which fold f x ~ fold g y. Use induction,
   i.e. prove that O(F)(X) ⊆ X, which implies that μO(F) ⊆ X, i.e.
   fold f x ~ fold g y for all (x, y) ∈ μO(F) ⊇ dom(~_μF)².

∀ F, f ~ g : F τ → τ, x, y : μF.
  fold f x ~ fold g y
  ⇒ ∀ x', y' : μF.
     (x', y') ∈ O(F)(x, y) ⇒ fold f x' ~ fold g y'

⇔ Def fold, property of fix.

∀ F, f ~ g : F τ → τ, x, y : μF.
  fold f x ~ fold g y
  ⇒ ∀ x', y' : μF.
    (x', y') ∈ O(F)(x, y)
    ⇒ (f ∘ L(F) (fold f) ∘ out) x' ~ (g ∘ L(F) (fold g) ∘ out) y'

⇐ Def ~, f ~ g.

∀ F, f ~ g : F τ → τ, x, y : μF.
  fold f x ~ fold g y
  ⇒ ∀ x', y' : μF.
    (x', y') ∈ O(F)(x, y)
    ⇒ (L(F) (fold f) ∘ out) x' ~ (L(F) (fold g) ∘ out) y'

⇔ Def O(F), out, in isomorphisms.

∀ F, f ~ g : F τ → τ, x, y : μF.
  fold f x ~ fold g y
  ⇒ ∀ x', y' : F μF.
    (x', y') ∈ O'_F(F)(x, y) ⇒ L(F) (fold f) x' ~ L(F) (fold g) y'

⇐ Generalise.

∀ F, f ~ g : F τ → τ, x, y : μF.
  fold f x ~ fold g y
  ⇒ ∀ G, x', y' : G μF.
    (x', y') ∈ O'_F(G)(x, y) ⇒ L(G) (fold f) x' ~ L(G) (fold g) y'

⇐ Induction over structure of G.
```

```
∀ F, f ∼ g : F τ → τ, x, y : μF.
  fold f x ∼ fold g y
  ⇒ ∀ G.
      (∀ G' < G , x', y' : G' μF.
        (x', y') ∈ O'_F(G')(x, y) ⇒ L(G') (fold f) x' ∼ L(G') (fold g) y')
     ⇒ ∀ x', y' : G μF.
         (x', y') ∈ O'_F(G)(x, y) ⇒ L(G) (fold f) x' ∼ L(G) (fold g) y'

⇐ Case over G.

○ G = Id.

  ⇔ Def L(Id), O'_F(Id), nothing < Id.

  ∀ F, f ∼ g : F τ → τ, x, y : μF.
    fold f x ∼ fold g y
    ⇒ ∀ x', y' : μF.
        (x', y') ∈ { (x, y) } ⇒ (fold f) x' ∼ (fold g) y'

  ⇔ Assumption. (Top-level inductive hypothesis.)

  ⊤

○ G = K_σ:

  ⇔ Def L(K_σ), O'_F(K_σ), nothing < K_σ.

  ∀ f ∼ g : F τ → τ, x, y : μF.
    fold f x ∼ fold g y
    ⇒ ∀ x', y' : σ.
        (x', y') ∈ { (a, b) | a, b ∈ dom(∼_σ), a ∼ b } ⇒ x' ∼ y'

  ⇔ Assumption.

  ⊤

○ G = G_1 × G_2.

  ⇐ Def O'_F(G_1 × G_2),
     L(G_1 × G_2) f p = seq p (L(G_1) f (fst p), L(G_2) f (snd p)),
     and seq p = id when p ≠ ⊥.
```

```
∀ f ~ g : F τ → τ, x, y : μF.
  fold f x ~ fold g y
  ⇒ (∀ $a_1$, $b_1$ : $G_1$ μF. ($a_1$, $b_1$) ∈ O'_F($G_1$)(x, y)
       ⇒ L($G_1$) (fold f) $a_1$ ~ L($G_1$) (fold g) $b_1$
     ∧
     ∀ $a_2$, $b_2$ : $G_2$ μF. ($a_2$, $b_2$) ∈ O'_F($G_2$)(x, y)
       ⇒ L($G_2$) (fold f) $a_2$ ~ L($G_2$) (fold g) $b_2$
    )
  ⇒ ∀ x', y' : G μF.
      (x', y') ∈ { (($a_1$, $a_2$), ($b_1$, $b_2$)) | ($a_1$, $b_1$) ∈ O'_F($G_1$)(x, y),
                                                  ($a_2$, $b_2$) ∈ O'_F($G_2$)(x, y) }
    ⇒ (L($G_1$) (fold f) (fst x'), L($G_2$) (fold f) (snd x')) ~
      (L($G_1$) (fold g) (fst y'), L($G_2$) (fold g) (snd y'))

⇔ Assumption.

⊤

○ G = $G_1$ + $G_2$:

⇔ Def O'_F($G_1$ + $G_2$).

∀ f ~ g : F τ → τ, x, y : μF.
  fold f x ~ fold g y
  ⇒ (∀ $a_1$, $b_1$ : $G_1$ μF. ($a_1$, $b_1$) ∈ O'_F($G_1$)(x, y)
       ⇒ L($G_1$) (fold f) $a_1$ ~ L($G_1$) (fold g) $b_1$
     ∧
     ∀ $a_2$, $b_2$ : $G_2$ μF. ($a_2$, $b_2$) ∈ O'_F($G_2$)(x, y)
       ⇒ L($G_2$) (fold f) $a_2$ ~ L($G_2$) (fold g) $b_2$
    )
  ⇒ ∀ x', y' : G μF.
      (x', y') ∈ { (inl(x'), inl(y')) | (x', y') ∈ O'_F($G_1$)(x, y) } ∪
                 { (inr(x'), inr(y')) | (x', y') ∈ O'_F($G_2$)(x, y) }
    ⇒ L(G) (fold f) x' ~ L(G) (fold g) y'

⇔ Def L($G_1$ + $G_2$).
```

```
  ∀ f ~ g : F τ → τ, x, y : μF.
    fold f x ~ fold g y
    ⇒ (∀ $a_1$, $b_1$ : $G_1$ μF. ($a_1$, $b_1$) ∈ O'_F($G_1$)(x, y)
          ⇒ L($G_1$) (fold f) $a_1$ ~ L($G_1$) (fold g) $b_1$
        ∧
        ∀ $a_2$, $b_2$ : $G_2$ μF. ($a_2$, $b_2$) ∈ O'_F($G_2$)(x, y)
          ⇒ L($G_2$) (fold f) $a_2$ ~ L($G_2$) (fold g) $b_2$
       )
    ⇒ ∀ x', y' : G μF.
         (x', y') ∈ O'_F($G_1$)(x, y)
           ⇒ inl(L($G_1$) (fold f) x') ~ inl(L($G_1$) (fold g) y')
         ∧
         (x', y') ∈ O'_F($G_2$)(x, y)
           ⇒ inr(L($G_2$) (fold f) x') ~ inr(L($G_2$) (fold g) y')

  ⇔ Assumption, def ~.

  ⊤
```

- unfold:

```
F: Polynomial functor on CPO.
unfold_F = λf. fix (λg. in ∘ L(F) g ∘ f)

∀ F. unfold ~ unfold

⇔ Def ~.

∀ F, f ~ g : τ → F τ, x ~ y : τ. unfold f x ~ unfold g y

⇐ Given F, f, g, let X ⊆ ℘($[\![\nu F]\!]^2$) be
      { (unfold f x', unfold g y') | x', y' : τ, x' ~ y' }.
   Use coinduction, i.e. prove that X ⊆ O(F)(X), which implies that
   X ⊆ νO(F), i.e. (unfold f x, unfold g y) ∈ νO(F) for all x ~ y : τ.

∀ F, f ~ g : τ → F τ, x ~ y : τ.
  (unfold f x, unfold g y) ∈
  O(F)({ (unfold f x', unfold g y') | x', y' : τ, x' ~ y' })

⇔ Def O(F).

∀ F, f ~ g : τ → F τ, x ~ y : τ.
  (unfold f x, unfold g y) ∈
    { (in(a), in(b)) | (a, b) ∈ O'_F(F)({ (unfold f x', unfold g y')
                                          | x', y' : τ, x' ~ y' }) }

⇔ Def unfold, property of fix.
```

```
∀ F, f ∼ g : τ → F τ, x ∼ y : τ.
  (in (L(F) (unfold f) (f x)), in (L(F) (unfold g) (g y)))
  ∈ { (in(a), in(b)) | (a, b) ∈ O'_F(F)({ (unfold f x', unfold g y')
                                          | x', y' : τ, x' ∼ y' }) }

⇐ Rewrite. (Note that the part below is slightly stronger than the
  one above for purely historical reasons...)

∀ F, f ∼ g : τ → F τ, x ∼ y : τ.
  ∃ x' ∼ y' : τ.
    (L(F) (unfold f) (f x), L(F) (unfold g) (g y))
    ∈ O'_F(F)(unfold f x', unfold g y')

⇐ Generalise.

∀ F, f ∼ g : τ → F τ, x ∼ y : τ.
  ∃ x' ∼ y' : τ.
  ∀ G.
    ∀ f' ∼ g' : τ → G τ.
      (L(G) (unfold f) (f' x), L(G) (unfold g) (g' y))
      ∈ O'_F(G)(unfold f x', unfold g y')

⇐ Induction over structure of G.

∀ F, f ∼ g : τ → F τ, x ∼ y : τ.
  ∃ x' ∼ y' : τ.
  ∀ G.
    ∀ G' < G.
      ∀ f' ∼ g' : τ → G' τ.
        (L(G') (unfold f) (f' x), L(G') (unfold g) (g' y))
        ∈ O'_F(G')(unfold f x', unfold g y')
    ⇒
    ∀ f' ∼ g' : τ → G τ.
      (L(G) (unfold f) (f' x), L(G) (unfold g) (g' y))
      ∈ O'_F(G)(unfold f x', unfold g y')

⇐ Case over G.

○ G = Id:

  ⇔ Def L(Id), O'_F(Id), nothing < Id:

  ∀ F, f ∼ g : τ → F τ, x ∼ y : τ.
    ∃ x' ∼ y' : τ.
      ∀ f' ∼ g' : τ → τ.
        (unfold f (f' x), unfold g (g' y))
        ∈ { (unfold f x', unfold g y') }

  ⇔ f' x ∼ g' y : τ by definition of ∼. For the existential
      quantifier we choose x' = f' x, y' = g' y.
  ⊤

○ G = K_σ:

  ⇔ Def L(K_σ), O'_F(K_σ), nothing < K_σ:

  ∀ F, f ~ g : τ → F τ, x ~ y : τ.
    ∃ x' ~ y' : τ.
      ∀ f' ~ g' : τ → σ.
        (f' x, g' y) ∈ { (x″, y″) | x″, y″ : σ, x″ ~ y″ }

  ⇔ f' x ~ g' y by definition of ~. For the existential quantifier
      we can choose x' = x, y' = y.

  ⊤

○ G = $G_1$ × $G_2$:

  ⇔ Def L($G_1$ × $G_2$), O'_F($G_1$ × $G_2$),
    f' x ~ g' y : ($G_1$ × $G_2$) τ ⇒ f' x ≠ ⊥ ≠ g' y.

  ∀ F, f ~ g : τ → F τ, x ~ y : τ.
    ∃ x' ~ y' : τ.
      ∀ G' < G.
        ∀ f' ~ g' : τ → G' τ.
          (L(G') (unfold f) (f' x), L(G') (unfold g) (g' y))
          ∈ O'_F(G')(unfold f x', unfold g y')
      ⇒
      ∀ f' ~ g' : τ → G τ.
        ( ( L($G_1$) (unfold f) (fst (f' x))
          , L($G_2$) (unfold f) (snd (f' x)) )
        , ( L($G_1$) (unfold g) (fst (g' x))
          , L($G_2$) (unfold g) (snd (g' x)) ) )
        ∈ { (($a_1$, $b_1$), ($a_2$, $b_2$))
          | ($a_1$, $a_2$) ∈ O'_F($G_1$)(unfold f x', unfold g y'),
            ($b_1$, $b_2$) ∈ O'_F($G_2$)(unfold f x', unfold g y') }

  ⇐ Rewrite, specialise.
```

```
 ∀ F, f ~ g : τ → F τ, x ~ y : τ.
  ∃ x' ~ y' : τ.
   ( ∀ f' ~ g' : τ → G_1 τ.
       (L(G_1) (unfold f) (f' x), L(G_1) (unfold g) (g' y))
       ∈ O'_F(G_1)(unfold f x', unfold g y')
     ∧
     ∀ f' ~ g' : τ → G_2 τ.
       (L(G_2) (unfold f) (f' x), L(G_2) (unfold g) (g' y))
       ∈ O'_F(G_2)(unfold f x', unfold g y')
   )
   ⇒
   ∀ f' ~ g' : τ → G τ.
     ( ( L(G_1) (unfold f) (fst (f' x))
       , L(G_2) (unfold f) (snd (f' x)) )
     , ( L(G_1) (unfold g) (fst (g' x))
       , L(G_2) (unfold g) (snd (g' x)) ) )
     ∈ { ((a_1, b_1), (a_2, b_2))
       | (a_1, a_2) ∈ O'_F(G_1)(unfold f x', unfold g y'),
         (b_1, b_2) ∈ O'_F(G_2)(unfold f x', unfold g y') }

 ⇔ Assumption, fst ∘ f' ~ fst ∘ g' : τ → G_1 τ and
  snd ∘ f' ~ snd ∘ g' : τ → G_2 τ by def ~.

 ⊤

∘ G = G_1 + G_2:

 ⇔ Def O'_F(G_1 + G_2).

 ∀ F, f ~ g : τ → F τ, x ~ y : τ.
  ∃ x' ~ y' : τ.
   ∀ G' < G.
    ∀ f' ~ g' : τ → G' τ.
      (L(G') (unfold f) (f' x), L(G') (unfold g) (g' y))
      ∈ O'_F(G')(unfold f x', unfold g y')
   ⇒
   ∀ f' ~ g' : τ → G τ.
     (L(G) (unfold f) (f' x), L(G) (unfold g) (g' y))
     ∈ { (inl(x'), inl(y'))
       | (x', y') ∈ O'_F(G_1)(unfold f x', unfold g y') } ∪
       { (inr(x'), inr(y'))
       | (x', y') ∈ O'_F(G_2)(unfold f x', unfold g y') }

 ⇐ Two cases. f' x ~ g' y ⇒ both inl or both inr.

 ∘ f' x = inl(a), g' y = inl(b):

  ⇐ Def L(G_1 + G_2).
  ∀ F, f ~ g : τ → F τ, x ~ y : τ.
    ∃ x' ~ y' : τ.
      ∀ G' < G.
        ∀ f' ~ g' : τ → G' τ.
          (L(G') (unfold f) (f' x), L(G') (unfold g) (g' y))
          ∈ O'_F(G')(unfold f x', unfold g y')
      ⇒
      ∀ f' ~ g' : τ → G τ.
        ∃ a ~ b : $G_1$ τ. f' x = inl(a) ∧ g' y = inl(b)
        ⇒ (inl(L($G_1$) (unfold f) a), inl(L($G_1$) (unfold g) b))
          ∈ { (inl(x'), inl(y'))
            | (x', y') ∈ O'_F($G_1$)(unfold f x', unfold g y') }

  ⇐ Rewrite, specialise.

  ∀ F, f ~ g : τ → F τ, x ~ y : τ.
    ∃ x' ~ y' : τ.
      ∀ f' ~ g' : τ → $G_1$ τ.
        (L($G_1$) (unfold f) (f' x), L($G_1$) (unfold g) (g' y))
        ∈ O'_F($G_1$)(unfold f x', unfold g y')
      ⇒
      ∀ f'' ~ g'' : τ → G τ.
        ∃ a ~ b : $G_1$ τ. f'' x = inl(a) ∧ g'' y = inl(b)
        ⇒ (L($G_1$) (unfold f) a, L($G_1$) (unfold g) b)
          ∈ O'_F($G_1$)(unfold f x', unfold g y')

  ⇔ Assumption, choose f' = λx. a, g' = λy. b. f', g' : τ → $G_1$ τ,
    both continuous, and f' ~ g' since a ~ b.

  ⊤

○ f' x = inr(a), g' y = inr(b): Analogous.
```

10 The PER is monotone

The predicate ~_σ can be viewed as a function ~_σ : $[\![\sigma]\!]^2 \to$ 1_⊥:

```
~_σ (x, y) = { ⋆, x ~_σ y,
             { ⊥, otherwise.
```

This function will now be shown to be monotone.

First note that it is enough to prove monotonicity for one argument, since the relation is symmetric. We need to show the following:

  $x \sim y \land y \sqsubseteq y' \Rightarrow x \sim y'$.

Proof: By induction over the type structure.

1: Done.

$\sigma \to \tau$:

  Given: $f\ x \sim g\ y \sqsubseteq g'\ y \Rightarrow f\ x \sim g'\ y$
  $f \sim g \sqsubseteq g'$

  Need to prove: $f \sim g'$

  Take $x \sim y$. We have $f\ x \sim g\ y \sqsubseteq g'\ y$. Done.

$\sigma \times \tau$:

  Given: $x_1 \sim x_2 \sqsubseteq x_2' \Rightarrow x_1 \sim x_2'$
  $y_1 \sim y_2 \sqsubseteq y_2' \Rightarrow y_1 \sim y_2'$
  $p \sim q \sqsubseteq q'$

  Need to prove: $p \sim q'$

  Since $p \sim q$ we get that $p, q \neq \bot$ which implies that $p = (x_1, y_1)$ and $q = (x_2, y_2)$ for some values with $x_1 \sim x_2$, $y_1 \sim y_2$. Furthermore $q \sqsubseteq q'$, so $q' = (x_2', y_2')$ for some x_2', y_2' with $x_2 \sqsubseteq x_2'$, $y_2 \sqsubseteq y_2'$. By the inductive hypothesis we then get that $p \sim q'$.

$\sigma + \tau$:

  Given: $a \sim b \sqsubseteq b' \Rightarrow a \sim b'$
  $p \sim q \sqsubseteq q'$

  Need to prove: $p \sim q'$

  Since $p \sim q$ we get that $p, q \neq \bot$ which implies either that p = inl(a) and q = inl(b) or that p = inr(a) and q = inr(b), for some a, b with $a \sim b$. Furthermore $q \sqsubseteq q'$, so in the inl case we get q' =

```
  inl(b') for some b' with b ⊑ b'. By the inductive hypothesis we then
  get that p ∼ q'. The inr case is analogous.
```

μF:

```
    ∀ x, y, y' ∈ ⟦μF⟧. x ∼ y ∧ y ⊑ y' ⇒ x ∼ y'
  ⇐
    ∀ x, y, x', y' ∈ ⟦μF⟧. x ∼ y ∧ x ⊑ x' ∧ y ⊑ y' ⇒ x' ∼ y'

    { Use induction. Let
  ⇐ |   X = { (x, y) ∈ ⟦μF⟧² | ∀ x', y' ∈ ⟦μF⟧. x ⊑ x' ∧ y ⊑ y' ⇒ x' ∼ y' }.
    { We are done if we can show that O(F)(X) ⊆ X.

    ∀ (x, y) ∈ O(F)(X).
      ∀ x', y' ∈ ⟦μF⟧.
        x ⊑ x' ∧ y ⊑ y' ⇒ x' ∼ y'
  ⇔ { in, out isomorphisms, see also per-and-in-out. }
    ∀ (x, y) ∈ O'_F(F)(X).
      ∀ x', y' ∈ ⟦F μF⟧.
        x ⊑ x' ∧ y ⊑ y' ⇒ x' ∼ y'
  ⇐ { Generalise. }
    ∀ G ≤ F.
      ∀ (x, y) ∈ O'_F(G)(X).
        ∀ x', y' ∈ ⟦G μF⟧.
          x ⊑ x' ∧ y ⊑ y' ⇒ x' ∼ y'
  ⇔ { Induction over G. }
    ∀ G ≤ F.
      ( ∀ G' < G, (x, y) ∈ O'_F(G')(X).
          ∀ x', y' ∈ ⟦G' μF⟧.
            x ⊑ x' ∧ y ⊑ y' ⇒ x' ∼ y'
      )
      ⇒ ∀ (x, y) ∈ O'_F(G)(X).
          ∀ x', y' ∈ ⟦G μF⟧.
            x ⊑ x' ∧ y ⊑ y' ⇒ x' ∼ y'
  ⇔ { Case analysis. }
```

• G = Id:

```
        ∀ (x, y) ∈ X.
          ∀ x', y' ∈ ⟦μF⟧.
            x ⊑ x' ∧ y ⊑ y' ⇒ x' ∼ y'
      ⇔ { Definition of X. }
        ⊤
```

• G = K_σ ≤ F:

```
        ∀ x, y ∈ ⟦σ⟧.
          x ∼ y
          ⇒ ∀ x', y' ∈ ⟦σ⟧.
              x ⊑ x' ∧ y ⊑ y' ⇒ x' ∼ y'
```

```
    ⇔ { Outer inductive hypothesis, σ < μF. }
      ⊤
```

- G = G₁ × G₂:

```
      ( ∀ G' < G, (x, y) ∈ O'_F(G')(X).
          ∀ x', y' ∈ ⟦G' μF⟧.
            x ⊑ x' ∧ y ⊑ y' ⇒ x' ~ y'
      )
      ⇒ ∀ (x₁, y₁) ∈ O'_F(G₁)(X), (x₂, y₂) ∈ O'_F(G₂)(X).
          ∀ x₁', y₁' ∈ ⟦G₁ μF⟧, x₂', y₂' ∈ ⟦G₂ μF⟧.
            x₁ ⊑ x₁' ∧ x₂ ⊑ x₂' ∧ y₁ ⊑ y₁' ∧ y₂ ⊑ y₂'
            ⇒ x₁' ~ y₁' ∧ x₂' ~ y₂'
    ⇔
      ⊤
```

- G = G₁ + G₂:

```
      ( ∀ G' < G, (x, y) ∈ O'_F(G')(X).
          ∀ x', y' ∈ ⟦G' μF⟧.
            x ⊑ x' ∧ y ⊑ y' ⇒ x' ~ y'
      )
      ⇒   ∀ (x₁, y₁) ∈ O'_F(G₁)(X).
            ∀ x₁', y₁' ∈ ⟦G₁ μF⟧.
              x₁ ⊑ x₁' ∧ y₁ ⊑ y₁' ⇒ x₁' ~ y₁'
        ∧ ∀ (x₂, y₂) ∈ O'_F(G₂)(X).
            ∀ x₂', y₂' ∈ ⟦G₂ μF⟧.
              x₂ ⊑ x₂' ∧ y₂ ⊑ y₂' ⇒ x₂' ~ y₂'
    ⇔
      ⊤
```

```
νF:

    ∀ x, y, y' ∈ ⟦νF⟧. x ~ y ∧ y ⊑ y' ⇒ x ~ y'
  ⇐
    ∀ x, y, x', y' ∈ ⟦νF⟧. x ~ y ∧ x ⊑ x' ∧ y ⊑ y' ⇒ x' ~ y'

    { Use coinduction. Let
  ⇐ |   X = { (x', y') | x, y, x', y' ∈ ⟦νF⟧, x ~ y, x ⊑ x', y ⊑ y' }.
    { We are done if we can show that X ⊆ O(F)(X).

    ∀ x, y, x', y' ∈ ⟦νF⟧.
      x ~ y ∧ x ⊑ x' ∧ y ⊑ y'
      ⇒ (x', y') ∈ O(F)(X)
  ⇔ { in, out isomorphisms, see also per-and-in-out. }
    ∀ x, y, x', y' ∈ ⟦F νF⟧.
      x ~ y ∧ x ⊑ x' ∧ y ⊑ y'
      ⇒ (x', y') ∈ O'_F(F)(X)
  ⇐ { Generalise. }
  ∀ G ≤ F, x, y, x', y' ∈ ⟦G νF⟧.
    x ~ y ∧ x ⊑ x' ∧ y ⊑ y'
    ⇒ (x', y') ∈ O'_F(G)(X)
⇔ { Induction over G. }
  ∀ G ≤ F.
    ( ∀ G' < G, x, y, x', y' ∈ ⟦G' νF⟧.
        x ~ y ∧ x ⊑ x' ∧ y ⊑ y'
        ⇒ (x', y') ∈ O'_F(G')(X)
    )
    ⇒ ∀ x, y, x', y' ∈ ⟦G νF⟧.
        x ~ y ∧ x ⊑ x' ∧ y ⊑ y'
        ⇒ (x', y') ∈ O'_F(G)(X)
⇔ { Case analysis. }
```

- G = Id:

```
      ∀ x, y, x', y' ∈ ⟦νF⟧.
        x ~ y ∧ x ⊑ x' ∧ y ⊑ y'
        ⇒ (x', y') ∈ X
    ⇔ { Definition of X. }
      ⊤
```

- G = K_σ ≤ F:

```
      ∀ x, y, x', y' ∈ ⟦σ⟧.
        x ~ y ∧ x ⊑ x' ∧ y ⊑ y'
        ⇒ x' ~ y'
    ⇔ { Outer inductive hypothesis, σ < νF. }
      ⊤
```

- G = G_1 × G_2:

```
      ( ∀ G' < G, x, y, x', y' ∈ ⟦G' νF⟧.
          x ~ y ∧ x ⊑ x' ∧ y ⊑ y'
          ⇒ (x', y') ∈ O'_F(G')(X)
      )
      ⇒ ∀ x₁, y₁, x₁', y₁' ∈ ⟦G₁ νF⟧, x₂, y₂, x₂', y₂' ∈ ⟦G₂ νF⟧.
          x₁ ~ y₁  ∧ x₂ ~ y₂  ∧
          x₁ ⊑ x₁' ∧ y₁ ⊑ y₁' ∧ x₂ ⊑ x₂' ∧ y₂ ⊑ y₂'
          ⇒ (x₁', y₁') ∈ O'_F(G₁)(X) ∧ (x₂', y₂') ∈ O'_F(G₂)(X)
    ⇔
      ⊤
```

- $G = G_1 + G_2$:

```
    ( ∀ G’ < G, x, y, x’, y’ ∈ ⟦G’ νF⟧.
        x ~ y ∧ x ⊑ x’ ∧ y ⊑ y’
        ⇒ (x’, y’) ∈ O’_F(G’)(X)
    )
    ⇒   ∀ x₁, y₁, x₁’, y₁’ ∈ ⟦G₁ νF⟧.
          x₁ ~ y₁ ∧ x₁ ⊑ x₁’ ∧ y₁ ⊑ y₁’
          ⇒ (x₁’, y₁’) ∈ O’_F(G₁)(X)
      ∧ ∀ x₂, y₂, x₂’, y₂’ ∈ ⟦G₂ νF⟧.
          x₂ ~ y₂ ∧ x₂ ⊑ x₂’ ∧ y₂ ⊑ y₂’
          ⇒ (x₂’, y₂’) ∈ O’_F(G₂)(X)
  ⇔
    ⊤
```

When recursive types are not used the function ~_σ is also continuous, i.e. least upper bounds are preserved:

```
 ⨆_i(x_i ~ y_i) = ⨆_i(x_i) ~ ⨆_i(y_i)
```

Proof:

Without recursive types all CPOs are finite, so monotonicity implies continuity.

11 Sizes can be assigned to some values

Each value of μ-type can be assigned a size
———

Since ℕ is a set we can use fold at the type

  (F ℕ → ℕ) → ⟪μF⟫ → ℕ.

(Note that ⟪μF⟫ is the codomain of in, so the above is well-typed.)

```
size_μF : ⟪μF⟫ → ℕ
size_μF x = 1 + fold size'_F x
  size'_G : G ℕ → ℕ
  size'_Id        s        = 1 + s
  size'_(K_τ)     x        = 1
  size'_($F_1$ × $F_2$) ($x_1$, $x_2$) = 1 + size'_$F_1$ $x_1$ + size'_$F_2$ $x_2$
  size'_($F_1$ + $F_2$) inl($x_1$)  = 1 + size'_$F_1$ $x_1$
  size'_($F_1$ + $F_2$) inr($x_2$)  = 1 + size'_$F_2$ $x_2$
```

  size' is well-defined since the size of the index functor always decreases.

```
size_⊥,μF : ⟨⟦μF⟧ → ℕ_⊥⟩
size_⊥,μF x = 1 + fold size'_⊥,F x
  size'_⊥,G : ⟨L(G) ℕ_⊥ → ℕ_⊥⟩
  size'_⊥,G         ⊥        = ⊥
  size'_⊥,Id        s        = 1 + s
  size'_⊥,(K_τ)     x        = 1
  size'_⊥,($F_1$ × $F_2$) ($x_1$, $x_2$) = 1 + size'_⊥,$F_1$ $x_1$ + size'_⊥,$F_2$ $x_2$
  size'_⊥,($F_1$ + $F_2$) inl($x_1$)  = 1 + size'_⊥,$F_1$ $x_1$
  size'_⊥,($F_1$ + $F_2$) inr($x_2$)  = 1 + size'_⊥,$F_2$ $x_2$
```

  size'_⊥ is well-defined since the size of the index functor always decreases.

  We assume that all arithmetical operations are strict. This implies that all functions above can be seen as arrows in CPO_⊥.

  Now we can prove that x ~ y ⇒ size_⊥ x = size_⊥ y ≠ ⊥. This implies that we can define a size for PER elements:

```
    size_~,μF : [~_μF] → ℕ
    size_~,μF [x] = size_⊥,μF x
```

  Proof:

    First define ~ for ℕ_⊥ (note that ℕ_⊥ ≠ μF for any F since we have too many liftings):

```
  m ~ n  ⇔  m = n ≠ ⊥.
```

This yields a PER.

We are done if we can show fold, (∘), (1+), size'_⊥,F ∈ dom(~).

1. fold:

   Here we have fold ∈ ⟨⟨L(F) ℕ_⊥ → ℕ_⊥⟩ → ⟨⟦μF⟧ → ℕ_⊥⟩⟩, for some polynomial functor F, with the definition

   ```
   fold = λf. fix (λg. f ∘ L(F) g ∘ out).
   ```

   The only difference between this definition and the previous, general definition of fold is that we have replaced a semantic domain ⟦σ⟧ with ℕ_⊥. Now, ℕ_⊥ cannot be represented as a semantic domain ⟦σ⟧. However, to prove that the definition above is well-formed and that fold is in dom(~) we can still use the proofs for the general fold. The reason is that the only property needed of ℕ_⊥ for the proofs to go through is that it is a CPO with a PER ~ defined on it. (Maybe it does not even need to be a CPO, but that is irrelevant.)

2. (∘): Easy.

3. (1+): Easy.

4. size'_⊥,F: Easy induction over F.

This gives rise to an inductive proof method: We can prove something by induction over the size of something of μ-type.

For simplicity we define the size of elements of type G μF as well:

```
size_(G μF)    x  = size'_G (G (fold size'_F) x)
size_⊥,(G μF)  x  = size'_⊥,G (G (fold size'_⊥,F) x)
size_~,(G μF) [x] = size_⊥,(G μF) x
```

We get a couple of easy lemmas, stated in general form, valid both for size and size_~:

```
size (in x) > size x
size (x, y) > size x
size (x, y) > size y
size inl(x) > size x
size inr(y) > size y
```

Note that the size does not decrease for μId. In fact it is not even well-defined. This is no problem, though, since ⟪μId⟫ = [~_μId] = ∅.

<u>Examples</u>

```
  size_μK_1 (in ⋆)
=
  1 + fold size'_K_1 (in ⋆)
=
  1 + size'_K_1 (K_1 (fold size'_K_1) ⋆)
=
  1 + size'_K_1 ⋆
=
  2

  size_μ(K_1 + Id) (in inr(in inl(⋆)))
=
  1 + fold size'_(K_1 + Id) (in inr(in inl(⋆)))
=
  1 + size'_(K_1 + Id) ((K_1 + Id) (fold size'_(K_1 + Id)) inr(in inl(⋆)))
=
  1 + size'_(K_1 + Id) (inr(fold size'_(K_1 + Id) (in inl(⋆))))
=
  2 + size'_Id (fold size'_(K_1 + Id) (in inl(⋆)))
=
  3 + fold size'_(K_1 + Id) (in inl(⋆))
=
  3 + size'_(K_1 + Id) ((K_1 + Id) (fold size'_(K_1 + Id)) inl(⋆))
=
  3 + size'_(K_1 + Id) inl(⋆)
=
  4 + size'_K_1 ⋆
=
  5
```

12 The approximation lemma

```
approx_⊥ is defined in
  [Graham Hutton and Jeremy Gibbons
   The Generic Approximation Lemma
   Information Processing Letters 79(4) p197-201, August 2001]:

approx_⊥ ∈ ℕ → ⟨⟦νF⟧ → ⟦νF⟧⟩
approx_⊥ 0     = λ_. ⊥
approx_⊥ (n+1) = in ∘ L(F) (approx_⊥ n) ∘ out

We have the approximation lemma:

  ∀ x, y ∈ ⟦νF⟧
    x = y  ⇔  ∀ n ∈ ℕ. approx_⊥ n x = approx_⊥ n y.

(Since we are working in CPO and all polynomial functors are locally
continuous.)
```

```
We can generalise the approximation lemma slightly. Define

  approx_⊥,G ∈ ℕ → ⟨⟦G νF⟧ → ⟦G νF⟧⟩
  approx_⊥,G n = L(G) (approx_⊥ n).

We get

  ∀ x, y ∈ ⟦G νF⟧
    x = y  ⇔  ∀ n ∈ ℕ. approx_⊥,G n x = approx_⊥,G n y.

Proof:

  ⇒: Trivial.

  ⇐: Done by easy induction over G.
```

□

13 Explicit characterisations of recursive type formers

Explicit characterisations of [∼_μF], [∼_νF], ⟪μF⟫ and ⟪νF⟫

Define the following monotone operators on the complete lattices
(℘([∼_μ/νF]), ⊆) and (℘(⟪μ/νF⟫), ⊆):

```
  S(F) : ℘([∼_μ/νF]) → ℘([∼_μ/νF])
  S(F)(X) = { in x | x ∈ S'_F(F)(X) }

  S'_F(G) : ℘([∼_τ]) → ℘([∼_(G τ)])
  S'_F(Id)(X)        = X
  S'_F(K_σ)(_)       = [∼_σ]
  S'_F(F_1 × F_2)(X) = { [(x, y)] | [x] ∈ S'_F(F_1)(X), [y] ∈ S'_F(F_2)(X) }
  S'_F(F_1 + F_2)(X) = { [inl(x)] | [x] ∈ S'_F(F_1)(X) } ∪
                       { [inr(y)] | [y] ∈ S'_F(F_2)(X) }

  Ŝ(F) : ℘(⟪μ/νF⟫) → ℘(⟪μ/νF⟫)
  Ŝ(F)(X) = { in x | x ∈ Ŝ'_F(F)(X) }

  Ŝ'_F(G) : ℘(⟪τ⟫) → ℘(⟪G τ⟫)
  Ŝ'_F(Id)(X)        = X
  Ŝ'_F(K_σ)(_)       = ⟪σ⟫
  Ŝ'_F(F_1 × F_2)(X) = Ŝ'_F(F_1)(X) × Ŝ'_F(F_2)(X)
  Ŝ'_F(F_1 + F_2)(X) = Ŝ'_F(F_1)(X) + Ŝ'_F(F_2)(X)
```

We will show that

```
  μS(F) = [∼_μF],
  μŜ(F) = ⟪μF⟫,
  νS(F) = [∼_νF], and
  νŜ(F) = ⟪νF⟫.
```

(Note that all fixpoints exist since the lattices are complete.)

---

1. μS(F) = [∼_μF]:

   - We know that μS(F) ⊆ [∼_μF] by construction (complete lattice).

   - μS(F) ⊇ [∼_μF]:

     ```
         ∀ x ∈ [∼_μF]. x ∈ μS(F)

       ⇐ { Generalise. }
     ```

```
  ∀ G, x ∈ [~_(G μF)]. x ∈ S'_F(G)(μS(F))

⇐ { Induction on size of x. }

  ∀ G, x ∈ [~_(G μF)].
    ∀ G', x' ∈ [~_(G' μF)].
      size_~,(G' μF) x' < size_~,(G μF) x
      ⇒ x' ∈ S'_F(G')(μS(F))
    ⇒ x ∈ S'_F(G)(μS(F))

⇔ { Case analysis on G. }

• G = Id:

  ⇔ { Definition S'_F(Id). }

    ∀ x ∈ [~_μF].
      ∀ G', x' ∈ [~_(G' μF)].
        size_~,(G' μF) x' < size_~,μF x
        ⇒ x' ∈ S'_F(G')(μS(F))
      ⇒ x ∈ μS(F)

  ⇔ { Fixpoint. }

    ∀ x ∈ [~_μF].
      ∀ G', x' ∈ [~_(G' μF)].
        size_~,(G' μF) x' < size_~,μF x
        ⇒ x' ∈ S'_F(G')(μS(F))
      ⇒ x ∈ S(F)(μS(F))

  ⇔ { Definition of S, in/out inverses. }

    ∀ x ∈ [~_μF].
      ∀ G', x' ∈ [~_(G' μF)].
        size_~,(G' μF) x' < size_~,μF x
        ⇒ x' ∈ S'_F(G')(μS(F))
      ⇒ out x ∈ S'_F(F)(μS(F))

  ⇔ { size_~,(F μF) (out x) < size_~,μF x (see size). }

    ⊤

• G = K_σ:

  ⇐ { Definition S'_F(K_σ), simplification. }

    ∀ x ∈ [~_σ].
      x ∈ [~_σ]

  ⇔ { Assumption. }
  ⊤
```

- `G = G₁ × G₂:`

```
⇔ { Definition S'_F(G₁ × G₂). }

  ∀ x ∈ [~_(G μF)].
    ∀ G', x' ∈ [~_(G' μF)].
      size_~,(G' μF) x' < size_~,(G μF) x
      ⇒ x' ∈ S'_F(G')(μS(F))
    ⇒ x ∈ { [(a, b)] | [a] ∈ S'_F(G₁)(μS(F)), [b] ∈ S'_F(G₂)(μS(F)) }

⇔ { Definition ~. }

  ∀ [(x, y)] ∈ [~_(G μF)].
    ∀ G', x' ∈ [~_(G' μF)].
      size_~,(G' μF) x' < size_~,(G μF) [(x, y)]
      ⇒ x' ∈ S'_F(G')(μS(F))
    ⇒ [x] ∈ S'_F(G₁)(μS(F)) ∧ [y] ∈ S'_F(G₂)(μS(F))

⇔ { size_~,(G₁ μF) [x] < size_~,(G μF) [(x, y)], and similarly
    for [y] (see size). }

  ⊤
```

- `G = G₁ + G₂:`

```
⇔ { Definition S'_F(G₁ + G₂). }

  ∀ x ∈ [~_(G μF)].
    ∀ G', x' ∈ [~_(G' μF)].
      size_~,(G' μF) x' < size_~,(G μF) x
      ⇒ x' ∈ S'_F(G')(μS(F))
    ⇒   x ∈ { [inl(a)] | [a] ∈ S'_F(G₁)(μS(F)) }
      ∨ x ∈ { [inr(b)] | [b] ∈ S'_F(G₂)(μS(F)) }

⇔ { We assume x = [inl(y)] for some y. The other case is
    analogous. }

  ∀ [y] ∈ [~_(G₁ μF)].
    ∀ G', x' ∈ [~_(G' μF)].
      size_~,(G' μF) x' < size_~,(G μF) [inl(y)]
      ⇒ x' ∈ S'_F(G')(μS(F))
    ⇒ [y] ∈ S'_F(G₁)(μS(F))

⇔ { size_~,(G₁ μF) [y] < size_~,(G μF) [inl(y)] (see size). }

  ⊤
```

□

2. `μŜ(F) = ⟪μF⟫:`

```
This proof is _almost_ a carbon copy of the previous one.
```

- `We know that μŜ(F) ⊆ ⟪μF⟫ by construction (complete lattice).`
- `μŜ(F) ⊇ ⟪μF⟫:`

```
    ∀ x ∈ ⟪μF⟫. x ∈ μŜ(F)

  ⇐ { Generalise. }

    ∀ G, x ∈ ⟪G μF⟫. x ∈ Ŝ'_F(G)(μŜ(F))

  ⇐ { Induction on size of x. }

    ∀ G, x ∈ ⟪G μF⟫.
      ∀ G', x' ∈ ⟪G' μF⟫.
        size_(G' μF) x' < size_(G μF) x
        ⇒ x' ∈ Ŝ'_F(G')(μŜ(F))
      ⇒ x ∈ Ŝ'_F(G)(μŜ(F))

  ⇔ { Case analysis on G. }

  • G = Id:

    ⇔ { Definition Ŝ'_F(Id). }

      ∀ x ∈ ⟪μF⟫.
        ∀ G', x' ∈ ⟪G' μF⟫.
          size_(G' μF) x' < size_μF x
          ⇒ x' ∈ Ŝ'_F(G')(μŜ(F))
        ⇒ x ∈ μŜ(F)

    ⇔ { Fixpoint. }

      ∀ x ∈ ⟪μF⟫.
        ∀ G', x' ∈ ⟪G' μF⟫.
          size_(G' μF) x' < size_μF x
          ⇒ x' ∈ Ŝ'_F(G')(μŜ(F))
        ⇒ x ∈ Ŝ(F)(μŜ(F))

    ⇔ { Definition of S, in/out inverses. }
    ∀ x ∈ ⟪μF⟫.
      ∀ G’, x’ ∈ ⟪G’ μF⟫.
        size_(G’ μF) x’ < size_μF x
        ⇒ x’ ∈ Ŝ’_F(G’)(μŜ(F))
      ⇒ out x ∈ Ŝ’_F(F)(μŜ(F))

  ⇔ { size_(F μF) (out x) < size_μF x (see size). }

    ⊤

• G = K_σ:

  ⇐ { Definition Ŝ’_F(K_σ), simplification. }

    ∀ x ∈ ⟪σ⟫.
      x ∈ ⟪σ⟫

  ⇔ { Assumption. }

    ⊤

• G = G_1 × G_2:

  ⇔ { Definition Ŝ’_F(G_1 × G_2). }

    ∀ x ∈ ⟪G μF⟫.
      ∀ G’, x’ ∈ ⟪G’ μF⟫.
        size_(G’ μF) x’ < size_(G μF) x
        ⇒ x’ ∈ Ŝ’_F(G’)(μŜ(F))
      ⇒ x ∈ { (a, b) | a ∈ Ŝ’_F(G_1)(μŜ(F)), b ∈ Ŝ’_F(G_2)(μŜ(F)) }

  ⇔ { Definition ⟪G μF⟫. }

    ∀ (x, y) ∈ ⟪G μF⟫.
      ∀ G’, x’ ∈ ⟪G’ μF⟫.
        size_(G’ μF) x’ < size_(G μF) (x, y)
        ⇒ x’ ∈ Ŝ’_F(G’)(μŜ(F))
      ⇒ x ∈ Ŝ’_F(G_1)(μŜ(F)) ∧ y ∈ Ŝ’_F(G_2)(μŜ(F))

  ⇔ { size_(G_1 μF) x < size_(G μF) (x, y), and similarly
      for y (see size). }

    ⊤

• G = G_1 + G_2:

  ⇔ { Definition Ŝ’_F(G_1 + G_2). }
  ∀ x ∈ ⟪G μF⟫.
    ∀ G', x' ∈ ⟪G' μF⟫.
      size_(G' μF) x' < size_(G μF) x
      ⇒ x' ∈ Ŝ'_F(G')(μŜ(F))
    ⇒   x ∈ { inl(a) | a ∈ Ŝ'_F(G₁)(μŜ(F)) }
      ∨ x ∈ { inr(b) | b ∈ Ŝ'_F(G₂)(μŜ(F)) }

⇔ { We assume x = inl(y) for some y. The other case is
    analogous. }

  ∀ y ∈ ⟪G₁ μF⟫.
    ∀ G', x' ∈ ⟪G' μF⟫.
      size_(G' μF) x' < size_(G μF) inl(y)
      ⇒ x' ∈ Ŝ'_F(G')(μŜ(F))
    ⇒ y ∈ Ŝ'_F(G₁)(μŜ(F))

⇔ { size_(G₁ μF) y < size_(G μF) inl(y) (see size). }

  ⊤
```

□

---

3. νS(F) = [~_νF]:

- We know that νS(F) ⊆ [~_νF] by construction (complete lattice).
- νS(F) ⊇ [~_νF]:

```
  [~_νF] ⊆ νS(F)

⇐ { Coinduction. }

  [~_νF] ⊆ S(F)([~_νF])

⇔ { Definition of S(F). }

  [~_νF] ⊆ { in x | x ∈ S'_F(F)([~_νF]) }

⇔ { in/out isomorphisms. }

  [~_(F νF)] ⊆ S'_F(F)([~_νF])

⇐ { Generalise. }

  ∀ G. [~_(G νF)] ⊆ S'_F(G)([~_νF])

⇐ { Induction. }
```

```
  ∀ G.
    ∀ G' < G. [~_(G' νF)] ⊆ S'_F(G')([~_νF])
    ⇒ [~_(G νF)] ⊆ S'_F(G)([~_νF])

⇔ { Case analysis. }

• G = Id:

  ⇔ { Simplify. }

    [~_νF] ⊆ [~_νF]

  ⇔

    ⊤

• G = K_σ:

  ⇔ { Simplify. }

    [~_σ] ⊆ [~_σ]

  ⇔

    ⊤

• G = G_1 × G_2:

  ⇔ { Simplify. }

    ∀ G' < G. [~_(G' νF)] ⊆ S'_F(G')([~_νF])
    ⇒   [~_(G_1 νF)] ⊆ S'_F(G_1)([~_νF])
      ∧ [~_(G_2 νF)] ⊆ S'_F(G_2)([~_νF])

  ⇔

    ⊤

• G = G_1 + G_2:

  ⇔ { Simplify. }

    ∀ G' < G. [~_(G' νF)] ⊆ S'_F(G')([~_νF])
    ⇒   [~_(G_1 νF)] ⊆ S'_F(G_1)([~_νF])
      ∧ [~_(G_2 νF)] ⊆ S'_F(G_2)([~_νF])

  ⇔

    ⊤
```

□

This proof is _almost_ a carbon copy of the previous one.

4. νŜ(F) = ⟪νF⟫:

   - We know that νŜ(F) ⊆ ⟪νF⟫ by construction (complete lattice).

   - νŜ(F) ⊇ ⟪νF⟫:

     ```
       ⟪νF⟫ ⊆ νŜ(F)

     ⇐ { Coinduction. }

       ⟪νF⟫ ⊆ Ŝ(F)(⟪νF⟫)

     ⇔ { Definition of Ŝ(F). }

       ⟪νF⟫ ⊆ { in x | x ∈ Ŝ'_F(F)(⟪νF⟫) }

     ⇔ { in/out isomorphisms. }

       ⟪F νF⟫ ⊆ Ŝ'_F(F)(⟪νF⟫)

     ⇐ { Generalise. }

       ∀ G. ⟪G νF⟫ ⊆ Ŝ'_F(G)(⟪νF⟫)

     ⇐ { Induction. }

       ∀ G.
         ∀ G' < G. ⟪G' νF⟫ ⊆ Ŝ'_F(G')(⟪νF⟫)
         ⇒ ⟪G νF⟫ ⊆ Ŝ'_F(G)(⟪νF⟫)

     ⇔ { Case analysis. }
     ```

     - G = Id:

       ```
       ⇔ { Simplify. }

         ⟪νF⟫ ⊆ ⟪νF⟫

       ⇔

         ⊤
       ```

     - G = K_σ:

       ```
       ⇔ { Simplify. }
       ```

```
  ⟪σ⟫ ⊆ ⟪σ⟫

⇔

  ⊤
```

- G = G_1 × G_2:

```
⇔ { Simplify. }

  ∀ G' < G. ⟪G' νF⟫ ⊆ Ŝ'_F(G')(⟪νF⟫)
  ⇒   ⟪G₁ νF⟫ ⊆ Ŝ'_F(G₁)(⟪νF⟫)
    ∧ ⟪G₂ νF⟫ ⊆ Ŝ'_F(G₂)(⟪νF⟫)

⇔

  ⊤
```

- G = G_1 + G_2:

```
⇔ { Simplify. }

  ∀ G' < G. ⟪G' νF⟫ ⊆ Ŝ'_F(G')(⟪νF⟫)
  ⇒   ⟪G₁ νF⟫ ⊆ Ŝ'_F(G₁)(⟪νF⟫)
    ∧ ⟪G₂ νF⟫ ⊆ Ŝ'_F(G₂)(⟪νF⟫)

⇔

  ⊤
```

□

14 A characterisation of the PER, valid for function-free types

```
Whenever σ does not contain any function spaces we have
  x ~_σ y  ⇔  x ∈ dom(~_σ) ∧ x = y.
```

```
Note first that the ⇐ case is easy, and x ~_σ y directly implies that
x ∈ dom(~_σ). For
  x ~_σ y ⇒ x = y
we use induction over the structure of σ.

1, +, ×: Easy.

μF:

    x ~_μF y  ⇒  x = y
  ⇐ { Induction. Let
    {   X = { (x, x) | x ∈ ⟦μF⟧ }.
    O(F)(X) ⊆ X
  ⇔
    ∀ (x, y) ∈ O(F)(X). x = y
  ⇔ { in, out bijections. }
    ∀ (x, y) ∈ O'_F(F)(X). x = y
  ⇐
    ∀ G ≤ F. (x, y) ∈ O'_F(G)(X). x = y
  ⇐ { Induction over G. }
    ∀ G ≤ F.
      ∀ G' < G, (x, y) ∈ O'_F(G')(X). x = y
      ⇒ ∀ (x, y) ∈ O'_F(G)(X). x = y
  ⇔ { Case analysis. }

  • G = Id: Trivial.
  • G = K_τ: By outer inductive hypothesis.
  • G = G₁ × G₂ or G₁ + G₂: By inner inductive hypothesis.

νF:

    ∀ x, y ∈ ⟦νF⟧. x ~_νF y  ⇒  x = y
  ⇔ { in, out bijections. }
    ∀ x, y ∈ ⟦F νF⟧. x ~_(F νF) y  ⇒  x = y
  ⇐ { Generalise. }
    ∀ G ≤ F, x, y ∈ ⟦G νF⟧. x ~_(G νF) y  ⇒  x = y
  ⇔ { The generalised approximation lemma. }
    ∀ G ≤ F, x, y ∈ ⟦G νF⟧. x ~_(G νF) y
      ⇒ ∀ n ∈ ℕ. approx_⊥,G n x = approx_⊥,G n y
  ⇔
    ∀ n ∈ ℕ, G ≤ F, x, y ∈ ⟦G νF⟧.
      x ~_(G νF) y ⇒ approx_⊥,G n x = approx_⊥,G n y
```

We proceed by lexicographic induction over first n and then G.

- n = 0, G = Id: Trivial.

- n = k+1, G = Id, x = in x', y = in y':

```
    approx_⊥,Id (k+1) (in x')
  =
    in (approx_⊥,F k x')
  = { Inductive hypothesis, x' ~ y'. }
    in (approx_⊥,F k y')
  =
    approx_⊥,Id (k+1) (in y')
```

- G = K_τ:

```
    approx_⊥,K_τ n x
  =
    x
  = { Outer inductive hypothesis. }
    y
  =
    approx_⊥,K_τ n y
```

- G = $G_1 \times G_2$, x = (x_1, x_2), y = (y_1, y_2):

```
    approx_⊥,(G₁ × G₂) n (x₁, x₂)
  =
    (approx_⊥,G₁ n x₁, approx_⊥,G₂ n x₂)
  = { Inductive hypothesis, x₁ ~ y₁, x₂ ~ y₂. }
    (approx_⊥,G₁ n y₁, approx_⊥,G₂ n y₂)
  =
    approx_⊥,(G₁ × G₂) n (y₁, y₂)
```

- G = G_1 + G_2, x = inl(x_1), y = inl(y_1), other case analogous.

```
    approx_⊥,(G₁ + G₂) n inl(x₁)
  =
    inl(approx_⊥,G₁ n x₁)
  = { Inductive hypothesis, x₁ ~ y₁. }
    inl(approx_⊥,G₁ n y₁)
  =
    approx_⊥,(G₁ + G₂) n inl(y₁)
```

(It may be nicer to use coinduction _for equality_ instead of the approximation lemma here. Indeed that could be true for the other proofs that use the approximation lemma as well. Maybe a general scheme for coinduction, that includes both ~ and =, could be set up.) □

15 The PER gives rise to a distributive bicartesian closed category

This part proves that the PER model gives rise to a bicartesian closed category.

Note: Originally it was proved that the category was also distributive (i.e. (σ × τ) + (σ × γ) ≅ σ × (τ + γ)), but apparently this follows since the category is bicartesian closed. See e.g. Huwig and Poigné, A note on inconsistencies caused by fixpoints in a cartesian closed category, Theoretical Computer Science 73(1), 101-112, 1990.

• It is a category

For lack of a better idea, name the category PER.

Objects: Types σ.

Morphisms: A morphism from σ to τ is an equivalence class of functions in dom(~_(σ → τ)).

Composition: [f] ∘ [g] = [f ∘ g] = [λv. f (g v)] (well-defined and associative).

Identity: id_σ = [id_σ] (a proper identity).

(Properties are inherited from the underlying structure.)

• Terminal object

The terminal object is 1 with [~_1] = {[⋆]}. (This object is apparently isomorphic to νId, since [~_νId] = {[⊥]}.)

We need to prove that there is exactly one morphism !_σ : σ → 1.

At least one: [⟦λx. ⋆⟧] = [λv. ⋆] : σ → 1.

At most one: Assume f : σ → 1. Then, for any x ∈ [~_σ] we have
f x = [⋆]. The result (f = [λv . ⋆]) follows by extensionality.

• Initial object

The initial object is μId with [~_μId] = ∅.

We need to prove that there is exactly one morphism <_σ : μId → σ.

At least one: $[\lambda v.\ \bot] : \mu Id \to \sigma$. (Note that $\lambda v.\ \bot \in dom(\sim)$ since it is non-bottom and $dom(\sim\_\mu Id) = \emptyset$.)

At most one: Assume $f : \mu Id \to \sigma$. Then, since $dom(\sim\_\mu Id) = \emptyset$, we have that $f = [\lambda v.\ \bot]$. Done.

• Products

The product construction is the product ($\times$) of the type system.

For the morphisms we have $[f] \vartriangle [g] = [\lambda v.\ (f\ v,\ g\ v)]$, $fst = [[\![fst]\!]]$, $snd = [[\![snd]\!]]$ (all well-defined).

Now, take arbitrary $f : \gamma \to \sigma$ and $g : \gamma \to \tau$. We have to show that
  $h = f \vartriangle g \Leftrightarrow fst \circ h = f \wedge snd \circ h = g$.

$\Rightarrow$: $fst \circ ([f] \vartriangle [g])$
   =
     $[[\![fst]\!] \circ (\lambda v.\ (f\ v,\ g\ v))]$
   =
     $[\lambda v.\ f\ v]$
   =
     $[f]$

   Similarly for $snd \circ h = g$.

$\Leftarrow$: Assume that $fst \circ [h] = [f]$ and $snd \circ [h] = [g]$. For an arbitrary v we get that
     $[fst\ (h\ v)] = [f\ v]$
   and
     $[snd\ (h\ v)] = [g\ v]$.
   Hence, since $\bot \notin dom(\sim)$ for product types,
     $[h\ v] = [(f\ v,\ g\ v)]$,
   i.e.
     $[h] = [\lambda v.\ (f\ v,\ g\ v)] = [f] \vartriangle [g]$.

• Coproducts

The coproduct construction is the sum (+) of the type system.

For the morphisms we have $[f] \triangledown [g] = [\lambda v.\ [\![case]\!]\ v\ f\ g]$, $inl = [[\![inl]\!]]$, $inr = [[\![inr]\!]]$ (all well-defined).

Now, take arbitrary $f : [\sim\_\sigma] \to [\sim\_\gamma]$ and $g : [\sim\_\tau] \to [\sim\_\gamma]$. We have to show that
  $h = f \triangledown g \Leftrightarrow h \circ inl = f \wedge h \circ inr = g$.

```
⇒:   ([f] ▽ [g]) ∘ inl
   =
     [(λv. ⟦case⟧ v f g) ∘ inl]
   =
     [λv. ⟦case⟧ inl(v) f g]
   =
     [λv. f v]
   =
     [f]

   Similarly for h ∘ inr = g.

⇐: Assume that h ∘ inl = f and h ∘ inr = g. For arbitrary vs we get
   that
     [h inl(v)] = [f v]
   and
     [h inr(v)] = [g v].
   Hence, since ⊥ ∉ dom(~) for product types,
     [h v] = [⟦case⟧ v f g],
   i.e.
     [h] = f ▽ g.
```

• Exponentials

The exponential σ^τ is τ → σ.

The corresponding morphisms are apply = [λ(f, x). f x] and curry = [λf x y. f (x, y)].

Actually we can easily derive curry from the universal property. (Note that we cannot assume that curry is a morphism. The expression curry f has to be defined for all f, but this only implies that curry is a total function from [~_((σ × τ) → γ)] to [~_(σ → (τ → γ))].)

```
  [g] = curry [f]  ⇔  apply ∘ ([g] × id) = [f]
⇔
  [g] = curry [f]  ⇔  [λ(f, x). f x] ∘ ([g] × id) = [f]
⇔ { As above. }
  [g] = curry [f]  ⇔  [λ(f, x). f x] ∘ [λ(x, y). (g x, y)] = [f]
⇔
  [g] = curry [f]  ⇔  [λ(x, y). g x y] = [f]
⇔
  [g] = curry [f]  ⇔  [g x y] = [f (x, y)]
⇔ { Extensionality. }
  (curry [f]) [x] [y] = [f (x, y)]
⇔
  curry [f] = [λx y. f (x, y)]
⇔ { Now we know that curry is a morphism. }
  curry = [λf x y. f (x, y)]
```

• Initial algebras

Given a polynomial functor F, we will prove that (μF, in) with in = [in] : F μF → μF is an initial F-algebra. That is, for any A, we have that fold_F = [⟦fold_F⟧] : (F A → A) → μF → A satisfies

```
  ∀ h : μF → A, f : F A → A.
    h = fold_F f  ⇔  h ∘ in = f ∘ F h.
```

Proof:

First notice that μF is a type, and that the morphisms above are actually morphisms of the correct types.

Note also that out = [out] is used below.

1. Show that

   ```
   h = fold_F f ⇒ h ∘ in = f ∘ F h,
   ```

   i.e. show that

   ```
   fold_F f ∘ in = f ∘ F (fold_F f)
   ```

   or equivalently ([in]/[out] are bijections since in/out are)

   ```
   fold_F f = f ∘ F (fold_F f) ∘ out.
   ```

   We can restate this as follows: For any [f], show that

   ```
   [⟦fold_F⟧ f] = [f ∘ L(F) (⟦fold_F⟧ f) ∘ out]
   ```

   (see functor-properties for proof that F = [L(F)]). By the definition of ⟦fold_F⟧ and fix we have that

   ```
   ⟦fold_F⟧ f = f ∘ L(F) (⟦fold_F⟧ f) ∘ out,
   ```

   and hence we are done (since ⟦fold_F⟧ f ∈ dom(~)).

2. Show that

   ```
   [h] ∘ in = [f] ∘ F [h] ⇒ [h] = fold_F [f],
   ```

   i.e.

   ```
   [h] = [f ∘ L(F) h ∘ out] ⇒ [h] = [⟦fold_F⟧ f],
   ```

   or, using extensionality,

```
 [h] = [f ∘ L(F) h ∘ out]
 ⇒ ∀ [x] ∈ [~_μF]. [h x] = [⟦fold_F⟧ f x].

Proof by induction over size of x (somewhat inspired by proof in
Meseguer and Goguen, Initiality, induction, and computability):

  ∀ [x] ∈ [~_μF].
    [h x] = [⟦fold_F⟧ f x]

⇔ { Properties of h and ⟦fold_F⟧. }

  ∀ [x] ∈ [~_μF].
    [(f ∘ L(F) h ∘ out) x] = [(f ∘ L(F) (⟦fold_F⟧ f) ∘ out) x]

⇐ { Generalise. }

  ∀ [x] ∈ [~_μF].
    [(L(F) h ∘ out) x] = [(L(F) (⟦fold_F⟧ f) ∘ out) x]

⇔ { in/out bijections. }

  ∀ [x] ∈ [~_(F μF)].
    [L(F) h x] = [L(F) (⟦fold_F⟧ f) x]

⇐ { Generalise. }

  ∀ G, [x] ∈ [~_(G μF)].
    [L(G) h x] = [L(G) (⟦fold_F⟧ f) x]

⇐ { Proof by induction over size of x. }

  ∀ G, [x] ∈ [~_(G μF)].
    ∀ G', [x'] ∈ [~_(G' μF)].
      size_~,(G' μF) [x'] < size_~,(G μF) [x]
      ⇒ [L(G') h x'] = [L(G') (⟦fold_F⟧ f) x']
    ⇒ [L(G) h x] = [L(G) (⟦fold_F⟧ f) x]

⇔ { Case analysis on G. }

• G = Id:

  ⇔ { Definition of Id, L(Id). }

    ∀ [x] ∈ [~_μF].
      ∀ G', [x'] ∈ [~_(G' μF)].
        size_~,(G' μF) [x'] < size_~,μF [x]
        ⇒ [L(G') h x'] = [L(G') (⟦fold_F⟧ f) x']
      [h x] = [⟦fold_F⟧ f x]
  ⇔ { Properties of h and ⟦fold_F⟧. }

    ∀ [x] ∈ [~_μF].
      ∀ G', [x'] ∈ [~_(G' μF)].
        size_~,(G' μF) [x'] < size_~,μF [x]
        ⇒ [L(G') h x'] = [L(G') (⟦fold_F⟧ f) x']
      [(f ∘ L(F) h ∘ out) x] = [(f ∘ L(F) (⟦fold_F⟧ f) ∘ out) x]

  ⇐ { Generalise. }

    ∀ [x] ∈ [~_μF].
      ∀ G', [x'] ∈ [~_(G' μF)].
        size_~,(G' μF) [x'] < size_~,μF [x]
        ⇒ [L(G') h x'] = [L(G') (⟦fold_F⟧ f) x']
      [L(F) h (out x)] = [L(F) (⟦fold_F⟧ f) (out x)]

  ⇔ { size_~,(F μF) [out x] < size_~,μF [x] (see size). }

    ⊤
```

- G = K_σ:

```
 ⇐ { Definition of K_σ, L(K_σ), simplification. }

   ∀ [x] ∈ [~_σ].
     ⇒ [x] = [x]

 ⇔ { Assumption. }

   ⊤
```

- G = G_1 × G_2:

```
 ⇔ { Definition of L(G), ~. }

   ∀ [(x₁, x₂)] ∈ [~_(G μF)].
     ∀ G', [x'] ∈ [~_(G' μF)].
       size_~,(G' μF) [x'] < size_~,μF [x]
       ⇒ [L(G') h x'] = [L(G') (⟦fold_F⟧ f) x']
     [L(G₁) h x₁] = [L(G₁) (⟦fold_F⟧ f) x₁] ∧
     [L(G₂) h x₂] = [L(G₂) (⟦fold_F⟧ f) x₂]

 ⇔ { size_~,(G₁ μF) [x₁] < size_~,(G μF) [(x₁, x₂)], and similarly
     for [x₂] (see size). }

   ⊤
```

- $G = G_1 + G_2$:

```
  ⇔ { We treat one case here (inl). The other one is analogous.
      Definition of L(G), ~. }

    ∀ [inl(x)] ∈ [~_(G μF)].
      ∀ G', [x'] ∈ [~_(G' μF)].
        size_~,(G' μF) [x'] < size_~,(G μF) [x]
        ⇒ [L(G') h x'] = [L(G') (⟦fold_F⟧ f) x']
      ⇒ [L(G₁) h x] = [L(G₁) (⟦fold_F⟧ f) x]

  ⇔ { size_~,(G₁ μF) [x] < size_~,(G μF) [inl(x)] (see size). }

    ⊤
```

□

- Final coalgebras

Given a polynomial functor F, we will prove that (νF, out) with out = [out] : νF → F νF is a final F-coalgebra. That is, for any τ, we have that unfold_F = [⟦unfold_F⟧] : (τ → F τ) → τ → νF satisfies

```
  ∀ h : τ → νF, f : τ → F τ.
    h = unfold_F f  ⇔  out ∘ h = F h ∘ f.
```

Proof:

First notice that νF is a type, and that the morphisms above are actually morphisms of the correct types.

1. Show that

   ```
   h = unfold_F f ⇔ out ∘ h = F h ∘ f,
   ```

   i.e. show that

   ```
   out ∘ unfold_F f = F (unfold_F f) ∘ f,
   ```

   or equivalently ([in]/[out] are bijections since in/out are)

   ```
   unfold_F f = in ∘ F (unfold_F f) ∘ f.
   ```

   We can restate this as follows: For any [f], show that

   ```
   [⟦unfold_F⟧ f] = [in ∘ L(F) (⟦unfold_F⟧ f) ∘ f]
   ```

   (see functor-properties for proof that F = [L(F)]). By the definition of ⟦unfold_F⟧ and fix we have that

```
  ⟦unfold_F⟧ f = in ∘ L(F) (⟦unfold_F⟧ f) ∘ f,
```

and hence we are done (since ⟦unfold_F⟧ f ∈ dom(~)).

2. Show that

```
  out ∘ [h] = F [h] ∘ [f] ⇒ [h] = unfold_F [f],
```

i.e.

```
  [h] = [in ∘ L(F) h ∘ f] ⇒ [h] = [⟦unfold_F⟧ f],
```

or, using extensionality,

```
  [h] = [in ∘ L(F) h ∘ f]
  ⇒ ∀ [x] ∈ [~_τ]. [h x] = [⟦unfold_F⟧ f x].
```

Proof:

```
  ∀ [x] ∈ [~_τ]. [h x] = [⟦unfold_F⟧ f x]

  { Use coinduction. Let
⇐ |   X = { (h x, ⟦unfold_F⟧ f x) | x ∈ dom(~_τ) }.
  { We need to show that X ⊆ O(F)(X).

  ∀ x ∈ dom(~_τ). (h x, ⟦unfold_F⟧ f x) ∈ O(F)(X)

⇔ { Top-level assumption about h, property of ⟦unfold_F⟧. }

  ∀ x ∈ dom(~_τ).
    ((in ∘ L(F) h ∘ f) x, (in ∘ L(F) (⟦unfold_F⟧ f) ∘ f) x) ∈ O(F)(X)

⇔ { Definition of O(F), in/out bijections. }

  ∀ x ∈ dom(~_τ).
    ((L(F) h ∘ f) x, (L(F) (⟦unfold_F⟧ f) ∘ f) x) ∈ O'_F(F)(X)

⇐ { Generalise. }

  ∀ x ∈ dom(~_(F τ)).
    (L(F) h x, L(F) (⟦unfold_F⟧ f) x) ∈ O'_F(F)(X)

⇐ { Generalise. }

  ∀ G, x ∈ dom(~_(G τ)).
    (L(G) h x, L(G) (⟦unfold_F⟧ f) x) ∈ O'_F(G)(X)

⇐ { Induction over G. }
```

```
 ∀ G.
   ∀ G' < G, x ∈ dom(∼_(G' τ)).
     (L(G') h x, L(G') (⟦unfold_F⟧ f) x) ∈ O'_F(G')(X)
   ⇒ ∀ x ∈ dom(∼_(G τ)).
     (L(G) h x, L(G) (⟦unfold_F⟧ f) x) ∈ O'_F(G)(X)

⇔ { Case analysis on G. }

• G = Id:

   ∀ x ∈ dom(∼_τ).
     (h x, ⟦unfold_F⟧ f x) ∈ X

 ⇔ { Definition of X. }

   ⊤

• G = K_σ:

   ∀ x ∈ dom(∼_σ).
     (x, x) ∈ { (x, y) | x, y ∈ dom(∼_σ), x ∼ y }

 ⇔ { Assumption. }

   ⊤

• G = G₁ × G₂:

   ∀ G' < G, x ∈ dom(∼_(G' τ)).
     (L(G') h x, L(G') (⟦unfold_F⟧ f) x) ∈ O'_F(G')(X)
   ⇒ ∀ x ∈ dom(∼_(G τ)).
     (L(G) h x, L(G) (⟦unfold_F⟧ f) x) ∈ O'_F(G)(X)

 ⇔ { Definition of L(G) and O'_F(G). }

   ∀ G' < G, x ∈ dom(∼_(G' τ)).
     (L(G') h x, L(G') (⟦unfold_F⟧ f) x) ∈ O'_F(G')(X)
   ⇒ ∀ x₁ ∈ dom(∼_(G₁ τ)), x₂ ∈ dom(∼_(G₂ τ)).
     (L(G₁) h x₁, L(G₁) (⟦unfold_F⟧ f) x₁) ∈ O'_F(G₁)(X) ∧
     (L(G₂) h x₂, L(G₂) (⟦unfold_F⟧ f) x₂) ∈ O'_F(G₂)(X)

 ⇔ { Assumption. }

   ⊤
```

- G = G_1 + G_2:

```
    ∀ G' < G, x ∈ dom(~_(G' τ)).
      (L(G') h x, L(G') (⟦unfold_F⟧ f) x) ∈ O'_F(G')(X)
    ⇒ ∀ x ∈ dom(~_(G τ)).
      (L(G) h x, L(G) (⟦unfold_F⟧ f) x) ∈ O'_F(G)(X)

  ⇔ { Definition of L(G) and O'_F(G). }

    ∀ G' < G, x ∈ dom(~_(G' τ)).
      (L(G') h x, L(G') (⟦unfold_F⟧ f) x) ∈ O'_F(G')(X)
    ⇒   ∀ x₁ ∈ dom(~_(G₁ τ)).
          (L(G₁) h x₁, L(G₁) (⟦unfold_F⟧ f) x₁) ∈ O'_F(G₁)(X)
      ∧ ∀ x₂ ∈ dom(~_(G₂ τ)).
          (L(G₂) h x₂, L(G₂) (⟦unfold_F⟧ f) x₂) ∈ O'_F(G₂)(X)

  ⇔ { Assumption. }

    ⊤
```

□

16 A characterisation of emptiness of domains of the PER

$\text{dom}(\sim_\sigma) = \emptyset \Leftrightarrow \langle\!\langle\sigma\rangle\!\rangle = \emptyset$

Proof by induction over the type structure:

- $\sigma = 1$: $\text{dom}(\sim_\sigma) \neq \emptyset \wedge \langle\!\langle\sigma\rangle\!\rangle \neq \emptyset$.

- $\sigma = \sigma_1 \times \sigma_2$:

```
  dom(~_σ) = ∅
⇔ { Definition of ~. }
  dom(~_σ₁) = ∅ ∨ dom(~_σ₂) = ∅
⇔ { Inductive hypothesis. }
  ⟪σ₁⟫ = ∅ ∨ ⟪σ₂⟫ = ∅
⇔
  ⟪σ⟫ = ∅
```

- $\sigma = \sigma_1 + \sigma_2$:

```
  dom(~_σ) = ∅
⇔ { Definition of ~. }
  dom(~_σ₁) = ∅ ∧ dom(~_σ₂) = ∅
⇔ { Inductive hypothesis. }
  ⟪σ₁⟫ = ∅ ∧ ⟪σ₂⟫ = ∅
⇔
  ⟪σ⟫ = ∅
```

- $\sigma = \sigma_1 \to \sigma_2$:

```
  dom(~_σ) = ∅
⇔ { Definition of ~. }
  { f ∈ ⟦σ₁ → σ₂⟧ | f ≠ ⊥, ∀ x, y ∈ ⟦σ₁⟧. x ~ y ⇒ f x ~ f y } = ∅
⇔
  { f ∈ ⟨⟦σ₁⟧ → ⟦σ₂⟧⟩ | ∀ x, y ∈ ⟦σ₁⟧. x ~ y ⇒ f x ~ f y } = ∅
⇔
  ¬(∃ f ∈ ⟨⟦σ₁⟧ → ⟦σ₂⟧⟩. ∀ x, y ∈ ⟦σ₁⟧. x ~ y ⇒ f x ~ f y)

  { Assume ∃ f ∈ ⟨⟦σ₁⟧ → ⟦σ₂⟧⟩. ∀ x, y ∈ ⟦σ₁⟧. x ~ y ⇒ f x ~ f y.
  | Then either dom(~_σ₁) = ∅ or we must have dom(~_σ₂) ≠ ∅.
  |
⇔ | On the other hand, assume dom(~_σ₁) = ∅. Since ⟨⟦σ₁⟧ → ⟦σ₂⟧⟩ ≠ ∅
  | (take λv. ⊥, for instance), we immediately get what we want.
  |
  | Finally, assume dom(~_σ₂) ≠ ∅. Take z ∈ dom(~_σ₂) and let f =
  { λv. z. Done.
  ¬(dom(~_σ_1) = ∅ ∨ dom(~_σ_2) ≠ ∅)
⇔
  dom(~_σ_1) ≠ ∅ ∧ dom(~_σ_2) = ∅
⇔ { Inductive hypothesis. }
  ⟪σ_1⟫ ≠ ∅ ∧ ⟪σ_2⟫ = ∅
⇔
  ⟪σ⟫ = ∅
```

- σ = μF:

```
  dom(~_σ) = ∅ ⇔ ⟪σ⟫ = ∅
⇔ { Definition of ~. }
  μO(F) = ∅ ⇔ ⟪μF⟫ = ∅
⇔ { See explicit-characterisations. }
  μO(F) = ∅ ⇔ μŜ(F) = ∅
⇔ { Induction, plus the fact that ∅ is a prefix point on both sides
  { above.
  O(F)(∅) ⊆ ∅ ⇔ Ŝ(F)(∅) ⊆ ∅
⇔
  O(F)(∅) = ∅ ⇔ Ŝ(F)(∅) = ∅
⇔
  O'_F(F)(∅) = ∅ ⇔ Ŝ'_F(F)(∅) = ∅
⇐ { Generalise. }
  ∀ G ≤ F. O'_F(G)(∅) = ∅ ⇔ Ŝ'_F(G)(∅) = ∅
⇐ { Induction. }
  ∀ G ≤ F.
    ∀ G' < G. O'_F(G')(∅) = ∅ ⇔ Ŝ'_F(G')(∅) = ∅
    ⇒ O'_F(G)(∅) = ∅ ⇔ Ŝ'_F(G)(∅) = ∅
⇔ { Case analysis. }
```

  - G = Id:

```
    O'_F(G)(∅) = ∅ ⇔ Ŝ'_F(G)(∅) = ∅
  ⇔
    ⊤
```

  - G = K_τ:

```
    O'_F(G)(∅) = ∅ ⇔ Ŝ'_F(G)(∅) = ∅
  ⇔
    ~_τ = ∅ ⇔ ⟪τ⟫ = ∅
  ⇔
    dom(~_τ) = ∅ ⇔ ⟪τ⟫ = ∅
  ⇔ { Outer inductive hypothesis, τ < μF since K_τ ≤ F. }
    ⊤
```

  - G = G_1 × G_2:

```
    O'_F(G)(∅) = ∅ ⇔ Ŝ'_F(G)(∅) = ∅
  ⇔
```

```
    (O'_F(G₁)(∅) = ∅ ∨ O'_F(G₂)(∅) = ∅) ⇔ (Ŝ'_F(G₁)(∅) = ∅ ∨ Ŝ'_F(G₂)(∅) = ∅)
  ⇔ { Inner inductive hypothesis. }
    ⊤
```

  - $G = G_1 + G_2$:

```
    O'_F(G)(∅) = ∅ ⇔ Ŝ'_F(G)(∅) = ∅
  ⇔
    (O'_F(G₁)(∅) = ∅ ∧ O'_F(G₂)(∅) = ∅) ⇔ (Ŝ'_F(G₁)(∅) = ∅ ∧ Ŝ'_F(G₂)(∅) = ∅)
  ⇔ { Inner inductive hypothesis. }
    ⊤
```

- σ = νF:

  First we define two subsets of the set of functors, using the following grammars:

```
  E ::= K_τ | E + E | E × F | F × E
        (For τ with ⟨⟨τ⟩⟩ = ∅.)

  NE ::= Id | K_τ | NE + F | F + NE | NE × NE
         (For τ with ⟨⟨τ⟩⟩ ≠ ∅.)
```

  The union of these subsets is the set of all functors. We can prove this by induction over the structure of a functor:

  - Id: Included in NE.

  - K_τ: Included in either E or NE, depending on whether ⟨⟨τ⟩⟩ = ∅ or not.

  - $F_1 \times F_2$: Inductively we know that F_1 and F_2 are both included in either E or NE. We get two cases:

    - One of F_1 and F_2 in E: $F_1 \times F_2$ in E.
    - F_1 and F_2 both in NE: $F_1 \times F_2$ in NE.

  - $F_1 + F_2$: Analogously.

  Now we will prove the following four statements, thereby establishing that dom(~_νF) = ∅ ⇔ ⟨⟨νF⟩⟩ = ∅:

  1. dom(~_νE) = ∅,
  2. ⟨⟨νE⟩⟩ = ∅,
  3. dom(~_νNE) ≠ ∅, and
  4. ⟨⟨νNE⟩⟩ ≠ ∅.

1. Note first that dom(∼_νE) = ∅ iff [∼_νE] = ∅. We have:

```
  [∼_νE]
= { See explicit-characterisations. }
  νS(E)
= { Fixpoint. }
  S(E)(νS(E))
= { Definition of S(E). }
  { [in x] | [x] ∈ S'_E(E)(νS(E)) }
```

   Note that { [in x] | [x] ∈ S'_E(E)(νS(E)) } = ∅ iff S'_E(E)(νS(E)) = ∅. We will now prove the generalised statement S'_E(G)(νS(E)) = ∅, for arbitrary G given by the grammar E, by induction over G:

   - G = K_τ, ⟪τ⟫ = ∅:

     ```
       S'_E(K_τ)(νS(E))
     =
       [∼_τ]
     = { Outer inductive hypothesis. }
       ∅
     ```

   - G = G_1 + G_2:

     ```
       S'_E(G_1 + G_2)(νS(E))
     =
       { [inl(x)] | [x] ∈ S'_E(G_1)(νS(E)) } ∪
       { [inr(y)] | [y] ∈ S'_E(G_2)(νS(E)) }
     = { Inductive hypothesis. }
       { [inl(x)] | [x] ∈ ∅ } ∪
       { [inr(y)] | [y] ∈ ∅ }
     =
       ∅
     ```

   - G = G_1 × F:

     ```
       S'_E(G_1 × F)(νS(E))
     =
       { [(x, y)] | [x] ∈ S'_E(G_1)(νS(E)), [y] ∈ S'_E(F)(νS(E)) }
     = { Inductive hypothesis. }
       { [(x, y)] | [x] ∈ ∅, [y] ∈ S'_E(F)(νS(E)) }
     =
       ∅
     ```

   - G = F × G_2:

     Similarly.

2. This proof follows the general structure of the proof of 1..

3. By the fundamental theorem we know that u $\in$ $[\![\nu F]\!]$ defined by

   u = $[\![$unfold (λx.y) $\star]\!]$ [y $\mapsto$ v]

   (with v $\in$ $[\![$F 1$]\!]$) is in dom(~_νF) if v $\in$ dom(~_(F 1)).

   We will now prove dom(~_νG) $\neq \emptyset$ by proving that dom(~_(G 1)) $\neq \emptyset$, for arbitrary G given by the grammar NE. The proof uses induction over G:

   - G = Id:

     $\star \in$ dom($\sim_1$)

   - G = K_τ, $\langle\!\langle\tau\rangle\!\rangle \neq \emptyset$:

     $\langle\!\langle\tau\rangle\!\rangle \neq \emptyset$ implies dom(~_τ) $\neq \emptyset$ by the outer inductive hypothesis.

   - G = G_1 $\times$ G_2:

     By the inductive hypothesis we know that there is a v_1 $\in$ dom(~_(G_1 1)) and a v_2 $\in$ dom(~_(G_2 1)). Hence (v_1, v_2) $\in$ dom(~_((G_1 $\times$ G_2) 1)).

   - G = G_1 + F:

     By the inductive hypothesis we know that there is a v $\in$ dom(~_(G_1 1)). Hence inl(v) $\in$ dom(~_((G_1 + F) 1)).

   - G = F + G_2:

     Similarly.

4. This proof follows the general structure of the proof of 3..

17 The partial surjective homomorphism

For all types σ:

```
∃ j_σ : ⟪σ⟫ ⇝ [~_σ].
  j_σ is surjective
  ∧
  j_(σ’ → σ) f and j_σ’ x both exist ⇒ j_σ (f x) = (j_(σ’ → σ) f) (j_σ’ x)
```

First note that, due to cardinality issues, we cannot find a bijective (total) function like the one above:

  All Scott domains have cardinality ≤ |℘(ℕ)| since
  1. Scott domains are ω-algebraic,
  2. an ω-algebraic domain has a countable set of basis elements, and
  3. every element of an ω-algebraic domain is the least upper bound of an ω-chain of basis elements.

  Now let Nat = μ(1 + Id). ⟦Nat⟧ is a Scott domain since S-DOM (the category of Scott domains and continuous functions) is closed under direct limits and 1 + Id is a locally continuous functor. (See Fokkinga and Meijer, Program Calculation Properties of Continuous Algebras for hints that ⟦Nat⟧ can actually be constructed using a direct limit.)

  ⟦(Nat → Nat) → Nat⟧ is also a Scott domain since S-DOM is closed under function spaces. Hence |⟦(Nat → Nat) → Nat⟧| ≤ |℘(ℕ)|. On the other hand, |⟪(Nat → Nat) → Nat⟫| = |(ℕ → ℕ) → ℕ| = |℘(℘(ℕ))| > |℘(ℕ)|. Hence j cannot be bijective for all types.

□

  (Domain theory results taken from Abbas Edalat’s lecture notes for the course "Domain Theory and Exact Computation" given 2002 at Imperial College.)

Second, note that j has to be partial:

  Take the total function isInfinite ∈ ⟪CoNat → Bool⟫, with CoNat = ν(1 + Id) and Bool = 1 + 1, defined by

```
                { True,  j n is defined and equal to [ω],
  isInfinite n = |
                { False, otherwise.
```

  (Here ω = ⟦unfold inr ⋆⟧ is the infinite "natural number", True = inl(⋆) and False = inr(⋆). Note that isInfinite takes j as an implicit parameter.)

Assume that we can find a function satisfying the specification of j, and assume that j_(CoNat → Bool) is total.

We get

(j isInfinite) (j n) = j (isInfinite n) = j True,

whenever j n = [ω] and whenever j n is defined but different from [ω] we get

(j isInfinite) (j n) = j False.

Notice that {j True} and {j False} are incomparable, since j is surjective and $\perp \notin$ dom(~_CoNat) (see troublesome-types). Notice also that {j True} and {j False} are well-defined. (The application of j is defined and the equivalence classes are singletons.)

Furthermore we have

{j True}

= { See above. Note that j is surjective. }

{j isInfinite} ω

= { Continuity. }

$\bigsqcup$_n {j isInfinite} $\text{inr}^n(\perp)$.

This implies that {j isInfinite} $\text{inr}^n(\perp) \sqsubseteq$ {j True} for all $n \in \mathbb{N}$, and furthermore {j isInfinite} $\text{inr}^n(\perp)$ = {j True} $\neq \perp$ for all $n \geq n_0$ for some $n_0 \in \mathbb{N}$.

By surjectivity of j we know that there is a subset $X \subseteq \langle\langle\text{CoNat}\rangle\rangle$ such that j n is defined for all $n \in X$ and

{ $[\text{inr}^n(\text{inl}(\star))]$ | $n \in \mathbb{N}$ } = { j n | $n \in X$ }.

This means that

(j isInfinite) (j n) = j False

for all $n \in X$, so we get

{j isInfinite} $\text{inr}^n(\text{inl}(\star))$ = {j False}

for all $n \in \mathbb{N}$, and hence by monotonicity

{j isInfinite} $\text{inr}^n(\perp) \sqsubseteq$ {j False},

```
  which contradicts what we derived above. Hence j_(CoNat → Bool)
  cannot be total. In particular, j isInfinite cannot be defined.
```
□

Third, if we add the requirements that
• j has to satisfy the main result (see main-result), and
• the definition of j for function spaces is the one given below,
then we get that j is not injective. Note that these requirements are
satisfied by the particular j defined below.

```
  Let isIsInf ∈ ⟪(CoNat → Bool) → Bool⟫ be defined by

                { True,  f = isInfinite,
    isIsInf f = │
                { False, otherwise.

  (Notice that the definition of isInfinite does not depend on any
  properties of j, not even that it is total.)

  Furthermore, let k = λf. False = ⟪λf. inr ⋆⟫.

  Assume now that j isIsInf and j k both exist. By surjectivity we
  have that for any f ∈ [~_(CoNat → Bool)] there is some f' ∈
  ⟪CoNat → Bool⟫ such that f = j f'. We get

    (j isIsInf) f = (j isIsInf) (j f') = j (isIsInf f').

  Since j isInfinite is not defined we get that (j isIsInf) f =
  j False for all f ∈ [~_(CoNat → Bool)]. The same is true for
  (j k) f. Hence, by extensionality, j isIsInf = j k, so j is not
  injective.

  So, are j isIsInf and j k both defined? By the requirement that j
  has to satisfy the main result, we get that j k has to be defined
  (note that k is closed). Furthermore j isIsInf is defined (and equal
  to j k) by the second requirement above since j isInfinite is not
  defined.
```
□

Thanks to Ross Paterson for the idea in the last proof above.

Now, on to the proof of the main statement:

By lexicographic induction over

1. the type structure, and

2. the size of the argument (only defined for μ-types).

We simultaneously construct a total right inverse j^{-1} to j. The function j^{-1} is then injective.

- σ = 1:

  j ⋆ = [⋆].

  j^{-1} [⋆] = ⋆.

  Surjective:

    j (j^{-1} [⋆])
  =
    j ⋆
  =
    [⋆].

- $\sigma = \tau_1 \times \tau_2$:

  j (x, y) = [({j x}, {j y})].

  j^{-1} [(x, y)] = (j^{-1} [x], j^{-1} [y]).

  Surjective:

    j (j^{-1} [(x, y)])
  =
    j (j^{-1} [x], j^{-1} [y])
  =
    [({j (j^{-1} [x])}, {j (j^{-1} [y])})]
  = { Inductive hypothesis. }
    [({[x]}, {[y]})]
  =
    [(x, y)].

- $\sigma = \tau_1 + \tau_2$:

  j inl(x) = [inl({j x})],
  j inr(y) = [inr({j y})].

  j^{-1} [inl(x)] = inl(j^{-1} [x]),
  j^{-1} [inr(y)] = inr(j^{-1} [y]).

  Surjective:

    j (j^{-1} [inl(x)])
  =
    j inl(j^{-1} [x])
  =
    [inl({j (j^{-1} [x])})]
  = { Inductive hypothesis. }

  [inl({[x]})]
=
  [inl(x)].

Other case analogous.

- $\sigma = \tau_1 \to \tau_2$:

  [Definition taken from:
     Harvey Friedman
     Equality between functionals
     Logic Colloquium
     Lecture Notes in Mathematics 453
     Springer-Verlag
     1975]

  Let j f be the element g $\in$ [~_($\tau_1 \to \tau_2$)] satisfying
    $\forall$ x $\in$ dom(j). g (j x) = j (f x) (which has to exist),
  if it (g) exists (in which case it is unique, see below).

  Uniqueness:
    Assume g_1, g_2 $\in$ [~_($\tau_1 \to \tau_2$)], both satisfying the condition above. Then $g_1 = g_2$ by extensionality (see definitions) since j_τ_1 is surjective (by inductive hypothesis).

  Surjectivity:
    Lemma:
      If $\langle\!\langle \tau_2 \rangle\!\rangle = \emptyset$, then [~_($\tau_1 \to \tau_2$)] = $\emptyset$ or $\langle\!\langle \tau_1 \rangle\!\rangle = \emptyset$.

      Proof:

      We have

        dom(~_($\tau_1 \to \tau_2$)) =
          { f $\in [\![\tau_1 \to \tau_2]\!]$ | f $\neq \bot$, $\forall$ x, y $\in [\![\tau_1]\!]$. x ~ y $\Rightarrow$ f x ~ f y }.

      Furthermore j_τ_2 is surjective, by inductive hypothesis, so $\langle\!\langle \tau_2 \rangle\!\rangle = \emptyset$ implies that dom(~_τ_2) = $\emptyset$. Thus f x ~ f y can never be true. Hence dom(~_($\tau_1 \to \tau_2$)) = $\emptyset$, unless dom(~_τ_1) = $\emptyset$ in which case we have dom(~_($\tau_1 \to \tau_2$)) = $[\![\tau_1 \to \tau_2]\!] \setminus \{ \bot \} \supseteq \{ \lambda v.\ \bot \} \neq \emptyset$. Finally we have that dom(~_τ_1) = $\emptyset$ implies that $\langle\!\langle \tau_1 \rangle\!\rangle = \emptyset$ (see per-empty).

    Three cases, based on the lemma:

    - $\langle\!\langle \tau_2 \rangle\!\rangle = \emptyset$, [~_($\tau_1 \to \tau_2$)] = $\emptyset$:

      j is trivially surjective (j = $\emptyset$, $j^{-1} = \emptyset$).

- $\langle\langle\tau_2\rangle\rangle = \emptyset$, $[\sim\_(\tau_1 \to \tau_2)] \neq \emptyset$, $\langle\langle\tau_1\rangle\rangle = \emptyset$:

  Take any element $g \in [\sim\_(\tau_1 \to \tau_2)]$. We have to show that g = j f for some $f \in \langle\langle\tau_1 \to \tau_2\rangle\rangle$. Note that $\langle\langle\tau_1\rangle\rangle = \langle\langle\tau_2\rangle\rangle = \emptyset$ implies that $\langle\langle\tau_1 \to \tau_2\rangle\rangle = \emptyset \to \emptyset = \{ \emptyset \}$. Let $f = \emptyset$. Now we have, for all $x \in dom(j\_\tau_1) = \emptyset$, that j (f x) = g (j x). Hence g is a (the) element satisfying the definition of j f.

  Note that, since all elements in $[\sim\_(\tau_1 \to \tau_2)]$ satisfy the definition of j f, we get that $[\sim\_(\tau_1 \to \tau_2)] = \{ g \}$. It follows that j^{-1}, defined by $j^{-1}\ g = \emptyset$, is a total right inverse to j.

- $\langle\langle\tau_2\rangle\rangle \neq \emptyset$:

  Take any element $g \in [\sim\_(\tau_1 \to \tau_2)]$. We have to show that g = j f for some $f \in \langle\langle\tau_1 \to \tau_2\rangle\rangle$.

  $$\text{Let } f\ x = \begin{cases} j^{-1}\ (g\ (j\ x)), & \text{if } x \in dom(j\_\tau_1),\\ y, & \text{otherwise,} \end{cases}$$

  where y is an arbitrary element in $\langle\langle\tau_2\rangle\rangle \neq \emptyset$. Furthermore we know inductively that j_τ_2 has a (total) right inverse $j^{-1} : [\sim\_\sigma] \to \langle\langle\sigma\rangle\rangle$, so f is well-defined.

  Now we have, for all $x \in dom(j)$,

  ```
    j (f x)

  = { x ∈ dom(j) }

    j (j^-1 (g (j x)))

  = { j^-1 is the right inverse of j }

    g (j x).
  ```

  Hence g is a (the) element satisfying the definition of j f.

  Note that this shows that

  $$j^{-1}\ g = \lambda x. \begin{cases} j^{-1}\ (g\ (j\ x)), & \text{if } x \in dom(j\_\tau_1),\\ y, & \text{otherwise,} \end{cases}$$

  with y as above, is a total right inverse to j.

Distributivity:
If j f and j x both exist, then j f (j x) = j (f x) by definition

(and hence j (f x) has to exist).

- σ = μF:

  ```
  j : ⟪μF⟫ ⇝ [∼_μF]
  j x = [in {J_F(F) (out x)}]
  ```

  ```
  J_F(G) : ⟪G μF⟫ ⇝ [∼_(G μF)]
  J_F(Id)         x      = j x
  J_F(K_σ)        x      = j x
  J_F($G_1$ × $G_2$) (x, y) = [({J_F($G_1$) x}, {J_F($G_2$) y})]
  J_F($G_1$ + $G_2$) inl(x) = [inl({J_F($G_1$) x})]
  J_F($G_1$ + $G_2$) inr(y) = [inr({J_F($G_2$) y})]
  ```

  j is well-defined (modulo partiality of j for the K_σ case), since

  1. all uses of j in J_F are either at a smaller type (K_σ), or apply to an argument which is smaller (Id, size defined in size), and

  2. recursive uses of J_F in the definition of J_F(G) only use J_F(G') for G' < G.

  ```
  $j^{-1}$ : [∼_μF] → ⟪μF⟫
  $j^{-1}$ x = in ($J^{-1}$_F(F) [out {x}])
  ```

  ```
  $J^{-1}$_F(G) : [∼_(G μF)] → ⟪G μF⟫
  $J^{-1}$_F(Id)         x        = $j^{-1}$ x
  $J^{-1}$_F(K_σ)        x        = $j^{-1}$ x
  $J^{-1}$_F($G_1$ × $G_2$) [(x, y)] = ($J^{-1}$_F($G_1$) [x], $J^{-1}$_F($G_2$) [y])
  $J^{-1}$_F($G_1$ + $G_2$) [inl(x)] = inl($J^{-1}$_F($G_1$) [x])
  $J^{-1}$_F($G_1$ + $G_2$) [inr(y)] = inr($J^{-1}$_F($G_2$) [y])
  ```

  j^{-1} is well-defined for the same reasons that j is (using size_∼ instead of size, see size), with no caveats regarding partiality.

  ```
    j ($j^{-1}$ [in x])
  =
    j (in ($J^{-1}$_F(F) [x]))
  =
    [in {J_F(F) ($J^{-1}$_F(F) [x])}]
  = { See lemma below. }
    [in x]
  ```

  Lemma: J_F(G) (J^{-1}_F(G) [x]) = [x], proved by induction over G ≤ F, with outer inductive hypothesis j ∘ j^{-1} = id (which is OK as long as we don't have F = Id, but then [∼_μF] = ∅ anyway, as noted in definitions).

- G = Id:

  ```
    J_F(Id) (J⁻¹_F(Id) [x])
  =
    j (j⁻¹ [x])
  = { Outer inductive hypothesis. }
    [x]
  ```

- G = K_σ:

  ```
    J_F(K_σ) (J⁻¹_F(K_σ) [x])
  =
    j (j⁻¹ [x])
  = { Outer inductive hypothesis (σ < νF). }
    [x]
  ```

- $G = G_1 \times G_2$:

  ```
    J_F(G₁ × G₂) (J⁻¹_F(G₁ × G₂) [(x, y)])
  =
    J_F(G₁ × G₂) (J⁻¹_F(G₁) [x], J⁻¹_F(G₂) [y])
  =
    [({J_F(G₁) (J⁻¹_F(G₁) [x])}, {J_F(G₂) (J⁻¹_F(G₂) [y])})]
  = { Inner inductive hypothesis. }
    [({[x]}, {[y]})]
  =
    [(x, y)]
  ```

- $G = G_1 + G_2$:

  ```
    J_F(G₁ + G₂) (J⁻¹_F(G₁ + G₂) [inl(x)])
  =
    J_F(G₁ + G₂) inl(J⁻¹_F(G₁) [x])
  =
    [inl({J_F(G₁) (J⁻¹_F(G₁) [x])})]
  = { Inner inductive hypothesis. }
    [inl({[x]})]
  =
    [inl(x)]
  ```

  Other case analogous.

- σ = νF:

To define j we'll use unfold in the category CPO:

```
  unfold : ⟨A → L(F) A⟩ → ⟨A → ⟦νF⟧⟩
```

Below we make use of the evaluation rules for unfold without mentioning it.

Let A = ⟪νF⟫_⊥, a flat CPO. All operations on ⟪·⟫ can be lifted to ⟪·⟫_⊥ by making them strict.

Let us first define j':

```
  j' ∈ ⟨⟪νF⟫_⊥ → ⟦νF⟧⟩
  j' = unfold (J'_νF(F) ∘ out)
```

Note that out ∈ ⟨⟪νF⟫_⊥ → ⟪F νF⟫_⊥⟩.

The helper function J'_σ is well-defined since recursive applications use a smaller functor. Note also that the instance of j used below (in j″) is at a smaller type. We also have to make sure that the function is continuous; since its domain is flat this follows directly from strictness.

```
  J'_σ(G) ∈ ⟨⟪G σ⟫_⊥ → L(G) ⟪σ⟫_⊥⟩
  J'_σ(G)       ⊥      = ⊥
  J'_σ(Id)      x      = x
  J'_σ(K_τ)     x      = j″ x
  J'_σ(G_1 × G_2) (x, y) = (J'_σ(G_1) x, J'_σ(G_2) y)
  J'_σ(G_1 + G_2) inl(x) = inl(J'_σ(G_1) x)
  J'_σ(G_1 + G_2) inr(y) = inr(J'_σ(G_2) y)

  j″ ∈ ⟨⟪σ⟫_⊥ → ⟦σ⟧⟩
  j″ ⊥ = ⊥
         { an arbitrary but fix element in j x, if x ∈ dom(j),
  j″ x = |
         { ⊥, otherwise
```

Note that in the proof below the function J'_F(G) defined by J'_F(G) = J'_νF(G) is used.

Given j' we can define j:

```
  j ∈ ⟪νF⟫ ⇝ [~_νF]
        { [j' x], if j' x ∈ dom(~),
  j x = |
        { undefined, otherwise.
```

We also need to define j^{-1}. To do this we use unfold in the category SET:

```
 unfold : (A → F A) → (A → ⟪νF⟫)
```

Let A = [~_νF], which of course is a set.

We can now define j^{-1}:

```
  j^-1 ∈ [∼_νF] → ⟪νF⟫
  j^-1 = unfold (J^-1_F(F) ∘ out')
```

Here we have

```
  out' ∈ [∼_νF] → [∼_(F νF)]
  out' [x] = [out x]
```

The helper function J^{-1}_F is well-defined since recursive applications use a smaller functor. Note also that the instance of j^{-1} used below is at a smaller type.

```
  J^-1_F(G) ∈ [∼_(G νF)] → G [∼_νF]
  J^-1_F(Id)      x        = x
  J^-1_F(K_σ)     x        = j^-1 x
  J^-1_F(G_1 × G_2) [(x, y)] = (J^-1_F(G_1) [x], J^-1_F(G_2) [y])
  J^-1_F(G_1 + G_2) [inl(x)] = inl(J^-1_F(G_1) [x])
  J^-1_F(G_1 + G_2) [inr(y)] = inr(J^-1_F(G_2) [y])
```

Now we have to prove that j^{-1} is the right inverse of j, thereby proving that j is surjective.

```
    ∀ x ∈ [∼_νF]. j (j^-1 x) = x
  ⇔
    ∀ x ∈ [∼_νF]. j' (j^-1 x) ∈ x
  ⇔
    ∀ x ∈ [∼_νF]. ∃ x' ∈ x. j' (j^-1 x) = x'
  ⇔
    ∀ x ∈ [∼_νF]. ∃ g ∈ [∼_νF] → ⟦νF⟧.
      g x ∈ x ∧ j' (j^-1 x) = g x
  ⇔ { OK even if [∼_νF] = ∅. }
    ∃ g ∈ [∼_νF] → ⟦νF⟧. ∀ x ∈ [∼_νF].
      g x ∈ x ∧ j' (j^-1 x) = g x
  ⇔ { in isomorphism. }
    ∃ g ∈ [∼_(F νF)] → ⟦F νF⟧. ∀ x ∈ [∼_(F νF)].
      g x ∈ x ∧ j' (j^-1 [in {x}]) = in (g x)

    {   j' (j^-1 [in {x}])
    |  =
    |   in (L(F) j' (J'_F(F) (F j^-1 (J^-1_F(F) x))))
  ⇔ |  =
    |   in (J' (J^-1 x))
    |
    { J' and J^-1 are defined below.

    ∃ g ∈ [∼_(F νF)] → ⟦F νF⟧. ∀ x ∈ [∼_(F νF)].
      g x ∈ x ∧ J' (J^-1 x) = g x
  ⇐ { Generalise. We have to limit G to be ≤ F here. See below. }
```

```
∀ G ≤ F.
  ∃ g ∈ [~_(G νF)] → ⟦G νF⟧.
    ∀ x ∈ [~_(G νF)].
      g x ∈ x ∧ J' (J⁻¹ x) = g x
```

Here we define

```
J⁻¹ ∈ [~_(G νF)] → ⟪G νF⟫
J⁻¹ = G j⁻¹ ∘ J⁻¹_F(G)
```

and

```
J' ∈ ⟨⟪G νF⟫_⊥ → ⟦G νF⟧⟩
J' = L(G) j' ∘ J'_F(G).
```

1. Find a g.

   We use unfold, just as above:

   ```
   g0 ∈ ⟨[~_νF]_⊥ → ⟦νF⟧⟩
   g0 = unfold (G_F ∘ out')

   G_G ∈ ⟨[~_(G νF)]_⊥ → L(G) [~_νF]_⊥⟩
   G_G ⊥                    = ⊥
   G_Id x                   = x
   G_(K_σ) x                = j″ (j⁻¹ x)
   G_(G1 × G2) [(x1, x2)] = (G_G1 [x1], G_G2 [x2])
   G_(G1 + G2) [inl(x1)]  = inl(G_G1 [x1])
   G_(G1 + G2) [inr(x2)]  = inr(G_G2 [x2])
   ```

   Since we can assume (inductively) that j″ $(j^{-1}$ x) is well-defined we get that g_0 and G_G satisfy their given type signatures with reasoning similar to the one above. (Note that we could not assume this if we hadn't assumed G ≤ F.)

   We can then define g as follows:

   ```
   g ∈ [~_(G νF)] → ⟦G νF⟧
   g = L(G) g0 ∘ G_G.
   ```

   Now one can easily check that g satisfies the following laws:

   ```
   Id:  g [in x0]     = in (g [x0])
   K_σ: g x           = j″ (j⁻¹ x)
   ×:   g [(x1, x2)] = (g [x1], g [x2])
   +:   g [inl(x1)]  = inl(g [x1])
   +:   g [inr(x2)]  = inr(g [x2])
   ```

2. We now have to show

```
  ∀ G ≤ F, x ∈ [~_(G νF)]. g x ∈ x.
```

We do this by induction over G. The case for Id follows below. The other cases are very similar to the respective inner cases inside the case for Id.

```
    ∀ x ∈ [~_νF]. g x ∈ x

  ⇐ { Use coinduction: Let X = { (g x, x') | x ∈ [~_νF], x' ∈ x }.
    { We are done if we can show that X ⊆ O(F)(X).

    ∀ x ∈ [~_νF], x' ∈ x. (g x, x') ∈ O(F)(X)
  ⇔ { See per-and-in-out and note that in is an isomorphism. }
    ∀ x ∈ [~_(F νF)], x' ∈ [in {x}]. (in (g x), x') ∈ O(F)(X)
  ⇔
    ∀ x ∈ [~_(F νF)], x' ∈ x. (g x, x') ∈ O'_F(F)(X)
  ⇐ { Generalise. }
    ∀ G ≤ F, x ∈ [~_(G νF)], x' ∈ x. (g x, x') ∈ O'_F(G)(X)
  ⇐ { Induction over G. }
    ∀ G ≤ F.
      ∀ G' < G. ∀ x ∈ [~_(G' νF)], x' ∈ x. (g x, x') ∈ O'_F(G')(X)
      ⇒ ∀ x ∈ [~_(G νF)], x' ∈ x. (g x, x') ∈ O'_F(G)(X)
  ⇔ { Case analysis. }
```

- G = Id:

```
      ∀ x ∈ [~_νF], x' ∈ x. (g x, x') ∈ X
    ⇔ { Definition of X. }
      ⊤
```

- G = K_σ:

```
      ∀ x ∈ [~_σ], x' ∈ x.
        (j″ (j⁻¹ x), x') ∈ { (x, y) | x, y ∈ dom(~_σ), x ~ y }

      { We have inductively (σ < νF) that
    ⇔ |   j (j⁻¹ x) = x,
      {  whereby j″ (j⁻¹ x) ∈ x.

      ⊤
```

- G = G_1 × G_2:

```
      ∀ G' < G. ∀ x ∈ [~_(G' νF)], x' ∈ x. (g x, x') ∈ O'_F(G')(X)
      ⇒ ∀ x ∈ [~_(G νF)], x' ∈ x. (g x, x') ∈ O'_F(G)(X)
    ⇔
      ∀ G' < G. ∀ x ∈ [~_(G' νF)], x' ∈ x. (g x, x') ∈ O'_F(G')(X)
      ⇒ ∀ x₁ ∈ [~_(G₁ νF)], x₂ ∈ [~_(G₂ νF)], x₁' ∈ x₁, x₂' ∈ x₂.
```

$((g\ x_1,\ g\ x_2),\ (x_1',\ x_2')) \in O'\_F(G)(X)$

$\Leftrightarrow$

$\forall\ G' < G.\ \forall\ x \in [\sim\_(G'\ \nu F)],\ x' \in x.\ (g\ x,\ x') \in O'\_F(G')(X)$
$\Rightarrow \forall\ x_1 \in [\sim\_(G_1\ \nu F)],\ x_2 \in [\sim\_(G_2\ \nu F)],\ x_1' \in x_1,\ x_2' \in x_2.$
$(g\ x_1,\ x_1') \in O'\_F(G_1)(X) \wedge (g\ x_2,\ x_2') \in O'\_F(G_2)(X)$

$\Leftrightarrow$

$\top$

- $G = G_1 + G_2$:

  $\forall\ G' < G.\ \forall\ x \in [\sim\_(G'\ \nu F)],\ x' \in x.\ (g\ x,\ x') \in O'\_F(G')(X)$
  $\Rightarrow \forall\ x \in [\sim\_(G\ \nu F)],\ x' \in x.\ (g\ x,\ x') \in O'\_F(G)(X)$

  $\Leftrightarrow$

  $\forall\ G' < G.\ \forall\ x \in [\sim\_(G'\ \nu F)],\ x' \in x.\ (g\ x,\ x') \in O'\_F(G')(X)$
  $\Rightarrow \forall\ x_1 \in [\sim\_(G_1\ \nu F)],\ x_1' \in x_1.\ (g\ x_1,\ x_1') \in O'\_F(G_1)(X)$
  $\wedge$
  $\forall\ x_2 \in [\sim\_(G_2\ \nu F)],\ x_2' \in x_2.\ (g\ x_2,\ x_2') \in O'\_F(G_2)(X)$

  $\Leftrightarrow$

  $\top$

3. Finally we have to show

   $\forall\ G \leq F,\ x \in [\sim\_(G\ \nu F)].\ J'\ (J^{-1}\ x) = g\ x.$

   We proceed by using the (generalised) approximation lemma,

   $\forall\ G \leq F,\ x \in [\sim\_(G\ \nu F)].\ J'\ (J^{-1}\ x) = g\ x$

   $\Leftrightarrow$

   $\forall\ G \leq F,\ x \in [\sim\_(G\ \nu F)],\ n \in \mathbb{N}.$
   $approx\_\bot,G\ n\ (J'\ (J^{-1}\ x)) = approx\_\bot,G\ n\ (g\ x)$

   $\Leftrightarrow$

   $\forall\ n \in \mathbb{N},\ G \leq F,\ x \in [\sim\_(G\ \nu F)].$
   $approx\_\bot,G\ n\ (J'\ (J^{-1}\ x)) = approx\_\bot,G\ n\ (g\ x)$

   and then lexicographic induction over first n and then G.

   - $n = 0,\ G = Id$:

     $approx\_\bot,Id\ 0\ (J'\ (J^{-1}\ x))$

     $=$

     $approx\_\bot\ 0\ (J'\ (J^{-1}\ x))$

     $=$

     $\bot$

     $=$

     $approx\_\bot\ 0\ (g\ [in\ x_0])$

     $=$

     $approx\_\bot,Id\ 0\ (g\ [in\ x_0])$

- n = k+1, G = Id, x = [in x_0]:

```
  approx_⊥,G (k+1) (J' (J^-1 x))
=
  approx_⊥ (k+1) (j' (j^-1 [in x_0]))
=
  approx_⊥ (k+1) (j' (in (F j^-1 (J^-1_F(F) (out' [in x_0])))))
=
  approx_⊥ (k+1) (j' (in (F j^-1 (J^-1_F(F) [x_0]))))
=
  approx_⊥ (k+1)
    (in (L(F) j' (J'_F(F) (out (in (F j^-1 (J^-1_F(F) [x_0])))))))
= { OK since in from SET and out lifted variant originally from SET. }
  approx_⊥ (k+1) (in (L(F) j' (J'_F(F) (F j^-1 (J^-1_F(F) [x_0])))))
=
  in (approx_⊥,F k (L(F) j' (J'_F(F) (F j^-1 (J^-1_F(F) [x_0])))))
=
  in (approx_⊥,F k (J' (J^-1 [x_0])))))
= { Outer inductive hypothesis. }
  in (approx_⊥,F k (g [x_0]))
=
  approx_⊥,G (k+1) (in (g [x_0]))
= { Property of g. }
  approx_⊥,G (k+1) (g [in x_0])
```

- G = K_σ:

```
  approx_⊥,G n (J' (J^-1 x))
=
  j'' (j^-1 x)
= { Property of g. }
  g x
=
  approx_⊥,G n (g x)
```

- G = G_1 × G_2, x = [(x_1, x_2)]:

```
  approx_⊥,G n (J' (J^-1 [(x_1, x_2)]))
=
  ( approx_⊥,G_1 n (J' (J^-1 [x_1]))
  , approx_⊥,G_2 n (J' (J^-1 [x_2]))
  )
= { Inductive hypothesis. }
  ( approx_⊥,G_1 n (g [x_1])
  , approx_⊥,G_2 n (g [x_2])
  )
=
  approx_⊥,G n (g [x_1], g [x_2])
= { Property of g. }
  approx_⊥,G n (g [(x_1, x_2)])
```

- G = G_1 + G_2, x = [inl(x_1)] (other case analogous):

```
  approx_⊥,G n (J' (J⁻¹ [inl(x₁)]))
=
  inl(approx_⊥,G₁ n (J' (J⁻¹ [x₁])))
= { Inductive hypothesis. }
  inl(approx_⊥,G₁ n (g [x₁]))
=
  approx_⊥,G n inl(g [x₁])
= { Property of g. }
  approx_⊥,G n (g [inl(x₁)])
```

□

18 Some properties satisfied by the partial surjective homomorphism

Various useful properties that j satisfies

Of course we have j ∘ j^{-1} = id, and whenever j f and j x are defined we have j f (j x) = j (f x).

If we restrict j_σ to j^{-1} (j ⟪σ⟫), then j_σ is total and j_σ and j^{-1}_σ are inverses:

1. The restriction of j to j^{-1} (j ⟪σ⟫) is total, since j (j^{-1} v) = v is always defined.

2. The restriction of j and j^{-1} are inverses:

   - ∀ v ∈ j ⟪σ⟫. j (j^{-1} v) = v
     by definition of j^{-1}.

   - ∀ v ∈ j^{-1} (j ⟪σ⟫). j^{-1} (j v) = v
     since v = j^{-1} v' for some v' and j (j^{-1} v') = v'.

When do we have ⟪t⟫ Γ ∈ j^{-1} (j ⟪·⟫)?

Probably not very often. Note that, for f ∈ dom(j),

```
    $j^{-1}$ (j f)
  = { For some y ∈ ⟪·⟫. }
    λv. {$j^{-1}$ (j (f v)), v ∈ dom(j)
        {y,                otherwise.
```

Since j^{-1} in general is not surjective we can not (in general) have j^{-1} (j f) = ⟪λx.x⟫ = λv.v. (And the arbitrary element y can not help; j^{-1} is not just one element away from being surjective.)

(Note that j^{-1} cannot be surjective due to cardinality issues.)

---

j id = [id]:

```
    [id] (j v)
  =
    j v
  =
    j (id v)
```

```
j ∘ = ∘: See below. Here we have defined ∘ = [∘].
```

```
j f ∘ j g = j (f ∘ g) whenever j f and j g both exist:

    (j f ∘ j g) (j v)
  =
    j f ((j g) (j v))
  =
    j f (j (g v))
  =
    j (f (g v))
  =
    j ((f ∘ g) v)
```

```
Note that the results above imply that j is a partial functor from

• the category of types and functions between the corresponding
  set-theoretic semantic domains

to

• the category PER defined in biccc.
```

```
Given g ∈ ⟪σ → τ⟫ ∩ dom(j) where ⊥ ∉ dom(~), we have
  j (F g) = [L(F) {j g}],
with both sides well-defined:

  Proof by induction over structure of F:

  (Properties from functor-properties silently used below.)

  • F = Id:

      [L(Id) {j g}]
    =
      [{j g}]
    =
      j g
    =
  j (Id g)
```

- F = K_σ:

  Note that this case requires that the left hand side is defined.

```
  [L(K_σ) {j g}]
=
  [id]
=
  j id
=
  j (K_σ g)
```

- F = $F_1 \times F_2$:

  For arbitrary v ∈ dom(j) we have:

```
  [L($F_1$ × $F_2$) {j g}] (j v)
=
  [L($F_1$ × $F_2$) {j g} {j v}]
= { ⊥ ∉ dom(∼) by assumption. }
  [(L($F_1$) {j g} {j v}, L($F_2$) {j g} {j v})]
=
  [({[L($F_1$) {j g} {j v}]}, {[L($F_2$) {j g} {j v}]})]
=
  [({[L($F_1$) {j g}] (j v)}, {[L($F_2$) {j g}] (j v)})]
= { Inductive hypothesis. }
  [({j ($F_1$ g) (j v)}, {j ($F_2$ g) (j v)})]
= { See above. }
  [({j ($F_1$ g v)}, {j ($F_2$ g v)})]
=
  j ($F_1$ g v, $F_2$ g v)
=
  j (($F_1$ × $F_2$) g v)
```

- F = $F_1 + F_2$:

  For arbitrary v ∈ dom(j) we have:

```
  [L($F_1$ + $F_2$) {j g}] (j v)
=
  [L($F_1$ + $F_2$) {j g} {j v}]
= { Assume v = inl(v'). Other case analogous. }
  [L($F_1$ + $F_2$) {j g} {j inl(v')}]
=
  [L($F_1$ + $F_2$) {j g} inl({j v'})]
=
  [inl(L($F_1$) {j g} {j v'})]
=
  [inl({[L(F₁) {j g} {j v'}]})]
=
  [inl({[L(F₁) {j g}] (j v')})]
= { Inductive hypothesis. }
  [inl({j (F₁ g) (j v')})]
= { See above. }
  [inl({j (F₁ g v')})]
=
  j inl(F₁ g v')
=
  j ((F₁ + F₂) g inl(v'))
= { By assumption above. }
  j ((F₁ + F₂) g v)
```

```
∀ v ∈ ⟪F μF⟫. j (in v) = [in {j v}]:

  Lemma: J_F(G) v = j v.

    Proof by induction over structure of G.

    G = Id or K_σ: OK.

    G = G₁ × G₂ or G₁ + G₂: OK (inductively).

  And then:

      j (in v)
    =
      [in {J_F(F) v}]
    =
      [in {j v}]
```

```
∀ v ∈ ⟪νF⟫. j out(v) = [out {j v}]:

  We reduce the out case to the in case:

    j out(v)
  = { Let v = in(v'). }
    j v'
  =
    [out {[in {j v'}]}]
  = { See below. }
    [out {j in(v')}]
  =
    [out {j v}]
```

Note that this in case isn't the same as the one above, though (different types). We get:

```
    j (in v)
  =
    [j' (in v)]
  =
    [in (L(F) j' (J'_F(F) v))]
  = { J' is defined in partial-surjective-homomorphism. }
    [in (J' v)]
  = { See next property. }
    [in {j v}]
```

For any v ∈ ⟪G νF⟫, if either side is well-defined, then [J' v] = j v. (J' is defined in partial-surjective-homomorphism.)

We prove this by induction over the structure of G.

- G = Id:

```
    j v
  =
    [j' v]
  =
    [J' v]
```

- G = K_σ:

```
    j v
  =
    [j″ v]
  =
    [J' v]
```

- G = G_1 × G_2:

```
    j (v₁, v₂)
  =
    [({j v₁}, {j v₂})]
  = { Inductive hypothesis. }
    [(J' v₁, J' v₂)]
  =
    [(L(G₁) j' (J'_F(G₁) v₁), L(G₂) j' (J'_F(G₂) v₂))]
  =
    [J' (v₁, v₂)]
```

- G = G_1 + G_2:

```
  j inl(v₁)
=
  [inl({j v₁})]
= { Inductive hypothesis. }
  [inl(J' v₁)]
=
  [inl(L(G₁) j' (J'_F(G₁) v₁))]
=
  [J' inl(v₁)]
```

  Other case analogous. □

(j⁻¹ f) (j⁻¹ x) = j⁻¹ (f x):

```
    (j⁻¹ f) (j⁻¹ x)
  = { Definition of j⁻¹. (Here y is an arbitrary element in ⟪·⟫.) }
    g (j⁻¹ x) where g v = {j⁻¹ (f (j v)), v ∈ dom(j)
                          {y,            otherwise
  =
    j⁻¹ (f (j (j⁻¹ x)))
  = { Right inverse. }
    j⁻¹ (f x)
```

---

j⁻¹ (f ∘ g) = j⁻¹ f ∘ j⁻¹ g:

```
    j⁻¹ (f ∘ g)
  = { Definition of j⁻¹. (Here y is an arbitrary element in ⟪·⟫.) }
    λv. {j⁻¹ ((f ∘ g) (j v)), v ∈ dom(j)
        {y,                   otherwise
  = { Definition ∘. }
    λv. {j⁻¹ (f (g (j v))), v ∈ dom(j)
        {y,                 otherwise
  = { Left inverse. }
    λv. {j⁻¹ (f (j (j⁻¹ (g (j v))))), v ∈ dom(j)
        {y,                           otherwise
  = { Definition j⁻¹, ∘. }
    j⁻¹ f ∘ j⁻¹ g
```

19 The main result

Main result, given that

- $t \in L_1$,
- seq is not used in t at a type with $\perp$ in its domain:

$$\forall\ x \in FV(t).\ \Gamma(x) \in dom(\sim) \land j\ \Gamma'(x) = [\Gamma(x)] \Rightarrow$$

$$j\ (\langle\!\langle t\rangle\!\rangle\ \Gamma') \text{ is well-defined} \land$$
$$j\ (\langle\!\langle t\rangle\!\rangle\ \Gamma') = [[\![t]\!]\ \Gamma]$$

Corollary (with analogous preconditions):

$$\forall\ x \in FV(t_1).\ \Gamma_1(x) \in dom(\sim) \land j\ \Gamma_1'(x) = [\Gamma_1(x)] \Rightarrow$$
$$\forall\ x \in FV(t_2).\ \Gamma_2(x) \in dom(\sim) \land j\ \Gamma_2'(x) = [\Gamma_2(x)] \Rightarrow$$

$$\langle\!\langle t_1\rangle\!\rangle\ \Gamma_1' = \langle\!\langle t_2\rangle\!\rangle\ \Gamma_2' \ \Rightarrow\ [\![t_1]\!]\ \Gamma_1 \sim [\![t_2]\!]\ \Gamma_2$$

Furthermore, if
$\langle\!\langle t_1\rangle\!\rangle\ \Gamma_1',\ \langle\!\langle t_2\rangle\!\rangle\ \Gamma_2' \in j^{-1}\ (j\ \langle\!\langle\cdot\rangle\!\rangle)$,
then the above implication is an equivalence.

The corollary follows immediately from the main result:

$\quad\langle\!\langle t_1\rangle\!\rangle\ \Gamma_1' = \langle\!\langle t_2\rangle\!\rangle\ \Gamma_2'$
$\Rightarrow$ { Extensionality. Note that both sides are well-defined. }
$\quad j\ (\langle\!\langle t_1\rangle\!\rangle\ \Gamma_1') = j\ (\langle\!\langle t_2\rangle\!\rangle\ \Gamma_2')$
$\Leftrightarrow$
$\quad[[\![t_1]\!]\ \Gamma_1] = [[\![t_2]\!]\ \Gamma_2]$
$\Leftrightarrow$
$\quad[\![t_1]\!]\ \Gamma_1 \sim [\![t_2]\!]\ \Gamma_2$

The "furthermore" part follows immediately from the corollary since j has a left inverse when restricted to $\{j^{-1}\ (j\ \langle\!\langle\sigma\rangle\!\rangle) \mid \sigma \text{ is a type}\}$ (see properties-of-j).

The main result is proved by induction over the structure of t, after noting that the fundamental theorem implies that $[\![t]\!]\ \Gamma \in dom(\sim)$.

t = x:

$\quad\quad[[\![x]\!]\ \Gamma]$
$\quad=$
$\quad\quad[\Gamma(x)]$
$\quad=$ { Assumption. }
$\quad\quad j\ \Gamma'(x)$
$\quad=$
$\quad\quad j\ (\langle\!\langle x\rangle\!\rangle\ \Gamma')$

```
t = t_1 t_2:

    [⟦t_1 t_2⟧ Γ]
  =
    ⟦t_1⟧ Γ (⟦t_2⟧ Γ)
  = { Inductive hypothesis, twice. }
    j (⟪t_1⟫ Γ') (j (⟪t_2⟫ Γ'))
  = { See partial-surjective-homomorphism. }
    j ((⟪t_1⟫ Γ') (⟪t_2⟫ Γ'))
  =
    j (⟪t_1 t_2⟫ Γ')

t = λx. t':

  Assuming t : σ → τ, pick an arbitrary v ∈ dom(j_σ). We have

    [⟦λx. t'⟧ Γ] (j v)
  =
    [λv. ⟦t'⟧ Γ[x ↦ v]] (j v)
  =
    [⟦t'⟧ Γ[x ↦ {j v}]]

  = { Inductive hypothesis;
    { note that [Γ[x ↦ {j v}](x)] = j v = j Γ'[x ↦ v].

    j (⟪t'⟫ Γ'[x ↦ v])
  =
    j ((λv. ⟪t'⟫ Γ'[x ↦ v]) v)
  =
    j ((⟪λx. t'⟫ Γ') v)

  Hence [⟦λx. t'⟧ Γ] satisfies the definition of j (⟪λx. t'⟫ Γ').

  (This proof method will be used without any explanations below.)

t = seq:

    [⟦seq⟧]
  =
    [f] where f b v = { ⊥, b = ⊥
                      { v, otherwise
  = { Assumption: ⊥ ∉ dom(~). }
    [λb v. v]
  = { See below. }
    j (λb v. v)
  =
    j ⟪seq⟫
    [λb v. v] (j b)
  =
    [id]
  = { See properties-of-j. }
    j id
  =
    j ((λb v. v) b)

t = ⋆:

    [⟦⋆⟧]
  =
    [⋆]
  =
    j ⟪⋆⟫

t = (,):

    [⟦(,)⟧]
  =
    [λx y. (x, y)]
  = { See below. }
    j (λx y. (x, y))
  =
    j ⟪(,)⟫

    [λx y. (x, y)] (j x) (j y)
  =
    [({j x}, {j y})]
  =
    j (x, y)
  =
    j ((λx y. (x, y)) x y)

t = fst:

    [⟦(,)⟧]
  =
    [f] where f p = { ⊥, p = ⊥
                    { x, p = (x, y)
  = { ⊥ ∉ dom(∼). }
    [λ(x, y). x]
  = { See below. }
    j (λ(x, y). x)
  =
    j ⟪fst⟫

    [λ(x, y). x] (j (x, y))
  =
    [λ(x, y). x] [({j x}, {j y})]
```

```
=
  j x
=
  j ((λ(x, y). x) (x, y))
```

- t = snd:

  Symmetrically.

- t = inl:

  ```
    [⟦inl⟧]
  =
    [λx. inl(x)]
  = { See below. }
    j (λx. inl(x))
  =
    j ⟪inl⟫

    [λx. inl(x)] (j x)
  =
    [inl({j x})]
  =
    j inl(x)
  =
    j ((λx. inl(x)) x)
  ```

- t = inr:

  Symmetrically.

- t = case:

  ```
    [⟦case⟧]
  =
                               {⊥,        v = ⊥
    [f] where f v $f_1$ $f_2$ = |$f_1$ $v_1$, v = inl($v_1$)
                               {$f_2$ $v_2$, v = inr($v_2$)
  = { ⊥ ∉ dom(~). }
    [f] where f v $f_1$ $f_2$ = {$f_1$ $v_1$, v = inl($v_1$)
                               {$f_2$ $v_2$, v = inr($v_2$)
  = { See below. }
    j ⟪case⟫
  ```

  Let us focus on the case when v = inl(v_1). The other case is analogous.

  ```
    [f] (j (inl($v_1$))) (j $f_1$) (j $f_2$)
      where f v $f_1$ $f_2$ = {$f_1$ $v_1$, v = inl($v_1$)
                             {$f_2$ $v_2$, v = inr($v_2$)
  ```

```
=
  [f] [inl({j v₁})] (j f₁) (j f₂)
    where f v f₁ f₂ = {f₁ v₁, v = inl(v₁)
                      {f₂ v₂, v = inr(v₂)
=
  [{j f₁} {j v₁}]
=
  [{j f₁}] [{j v₁}]
=
  j f₁ (j v₁)
= { See partial-surjective-homomorphism. }
  j (f₁ v₁)
=
  j (⟪case⟫ inl(v₁) f₁ f₂)
```

- t = in:

```
  [⟦in⟧]
=
  [λv. in(v)]
= { See below. }
  j (λv. in(v))
=
  j ⟪in⟫

  [λv. in(v)] (j v)
=
  [in({j v})]
= { See properties-of-j. }
  j in(v)
=
  j ((λv. in(v)) v)
```

- t = out:

Similarly.

- t = fold:

```
Note that
  ⟦fold⟧ f = f ∘ F (⟦fold⟧ f) ∘ out
and
  ⟪fold⟫ f = f ∘ F (⟪fold⟫ f) ∘ out.

We get

  [⟦fold⟧] = j ⟪fold⟫
⇔
  ∀ f ∈ dom(j_(F σ → σ)), v ∈ dom(j_μF).
    [⟦fold⟧] (j f) (j v) = j (⟪fold⟫ f v)
```

```
⇔
  ∀ f ∈ dom(j_(F σ → σ)), v ∈ dom(j_μF).
    ∀ f_0 ∈ j f, v_0 ∈ j v.
      [⟦fold⟧ f_0 v_0] = j (⟪fold⟫ f v)
⇐ { Generalise. }
  ∀ G, f ∈ dom(j_(F σ → σ)), v ∈ dom(j_(G μF)).
    ∀ f_0 ∈ j f, v_0 ∈ j v.
      [L(G) (⟦fold⟧ f_0) v_0] = j (G (⟪fold⟫ f) v)
⇐ { Induction on size of v. }
  ∀ G, f ∈ dom(j_(F σ → σ)), v ∈ dom(j_(G μF)).
    ∀ G', v' ∈ dom(j_(G' μF)).
      size_(G' μF) v' < size_(G μF) v
      ⇒ ∀ f_0 ∈ j f, v_0' ∈ j v'.
          [L(G') (⟦fold⟧ f_0) v_0'] = j (G' (⟪fold⟫ f) v')
    ⇒ ∀ f_0 ∈ j f, v_0 ∈ j v.
        [L(G) (⟦fold⟧ f_0) v_0] = j (G (⟪fold⟫ f) v)
⇔ { Case analysis. }
```

- G = Id:

```
  Here we have v = in v'. Let [in v_0'] = j (in v') = [in {j v'}].

    [L(G) (⟦fold⟧ f_0) (in v_0')]
  =
    [⟦fold⟧ f_0 (in v_0')]
  =
    [f_0 (L(F) (⟦fold⟧ f_0) v_0')]
  =
    [f_0 {[L(F) (⟦fold⟧ f_0) v_0']}]
  = { Inductive hypothesis: v' < v. }
    [f_0 {j (F (⟪fold⟫ f) v')}]
  =
    j f (j (F (⟪fold⟫ f) v'))
  =
    j (f (F (⟪fold⟫ f) v'))
  =
    j (⟪fold⟫ f (in v'))
  =
    j (G (⟪fold⟫ f) (in v'))
```

- G = K_τ:

```
    [L(G) (⟦fold⟧ f_0) v_0]
  =
    [v_0]
  =
    j v
  =
    j (G (⟪fold⟫ f) v)
```

  - $G = G_1 \times G_2$:

    Here we have $v = (v_1, v_2)$. Let $[(v_{0\,1}, v_{0\,2})] = j\ (v_1, v_2) = [(\{j\ v_1\}, \{j\ v_2\})]$.

    ```
      [L(G) ($[\![$fold$]\!]$ $f_0$) ($v_{0\,1}$, $v_{0\,2}$)]
    =
      [(L($G_1$) ($[\![$fold$]\!]$ $f_0$) $v_{0\,1}$, L($G_2$) ($[\![$fold$]\!]$ $f_0$) $v_{0\,2}$)]
    =
      [({[L($G_1$) ($[\![$fold$]\!]$ $f_0$) $v_{0\,1}$]}, {[L($G_2$) ($[\![$fold$]\!]$ $f_0$) $v_{0\,2}$]})]
    = { Inductive hypothesis: $v_1$ < v, $v_2$ < v. }
      [({j ($G_1$ ($\langle\!\langle$fold$\rangle\!\rangle$ f) $v_1$)}, {j ($G_2$ ($\langle\!\langle$fold$\rangle\!\rangle$ f) $v_2$)})]
    = { Definition of j. }
      j ($G_1$ ($\langle\!\langle$fold$\rangle\!\rangle$ f) $v_1$, $G_2$ ($\langle\!\langle$fold$\rangle\!\rangle$ f) $v_2$)
    =
      j (G ($\langle\!\langle$fold$\rangle\!\rangle$ f) ($v_1$, $v_2$))
    ```

  - $G = G_1 + G_2$:

    Here we have $v = inl(v_1)$ or $v = inr(v_2)$. The second case is omitted. Let $[inl(v_{0\,1})] = j\ inl(v_1) = [inl(\{j\ v_1\})]$.

    ```
      [L(G) ($[\![$fold$]\!]$ $f_0$) inl($v_{0\,1}$)]
    =
      [inl(L($G_1$) ($[\![$fold$]\!]$ $f_0$) $v_{0\,1}$)]
    =
      [inl({[L($G_1$) ($[\![$fold$]\!]$ $f_0$) $v_{0\,1}$]})]
    = { Inductive hypothesis: $v_1$ < v. }
      [inl({j ($G_1$ ($\langle\!\langle$fold$\rangle\!\rangle$ f) $v_1$)})]
    = { Definition of j. }
      j inl($G_1$ ($\langle\!\langle$fold$\rangle\!\rangle$ f) $v_1$)
    =
      j (G ($\langle\!\langle$fold$\rangle\!\rangle$ f) inl($v_1$))
    ```

- t = unfold:

  Note that
  $[\![\text{unfold}]\!]\ f = in \circ F\ ([\![\text{fold}]\!]\ f) \circ f$
  and
  $\langle\!\langle\text{unfold}\rangle\!\rangle\ f = in \circ F\ (\langle\!\langle\text{fold}\rangle\!\rangle\ f) \circ f$.

  We get:

  ```
    [$[\![$unfold$]\!]$] = j $\langle\!\langle$unfold$\rangle\!\rangle$
  ⇔
    ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_σ).
      [$[\![$unfold$]\!]$] (j f) (j v) = j ($\langle\!\langle$unfold$\rangle\!\rangle$ f v)
  ⇔
  ```

```
   ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_σ).
     ∀ f_0 ∈ j f, v_0 ∈ j v.
       [⟦unfold⟧ f_0 v_0] = j (⟪unfold⟫ f v)

   { Use coinduction. Let
   │
   │        { ( ⟦unfold⟧ f_0 v_0       │ f ∈ dom(j_(σ → F σ)), }
 ⇐ │    X = │ , j' (⟪unfold⟫ f v)  │ v ∈ dom(j_σ),         │.
   │        { )                    │ f_0 ∈ j f, v_0 ∈ j v    }
   │
   │ We are done if we can show that X ⊆ O(F)(X).
   { For definition of j', see partial-surjective-homomorphism.

   ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_σ).
     ∀ f_0 ∈ j f, v_0 ∈ j v.
       (⟦unfold⟧ f_0 v_0, j' (⟪unfold⟫ f v)) ∈ O(F)(X)
 ⇔ { out isomorphism. }
   ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_σ).
     ∀ f_0 ∈ j f, v_0 ∈ j v.
       (out (⟦unfold⟧ f_0 v_0), out (j' (⟪unfold⟫ f v))) ∈ O'_F(F)(X)

   {   out (⟦unfold⟧ f_0 v_0)
   │ =
   │   L(F) (⟦unfold⟧ f_0) (f_0 v_0)
   │
   │   out (j' (⟪unfold⟫ f v))
 ⇔ │ =
   │   L(F) j' (J'_F(F) (F (⟪unfold⟫ f) (f v)))
   │ =
   │   J' (F (⟪unfold⟫ f) (f v))
   │
   { For definition of J', see partial-surjective-homomorphism.

   ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_σ).
     ∀ f_0 ∈ j f, v_0 ∈ j v.
       (L(F) (⟦unfold⟧ f_0) (f_0 v_0), J' (F (⟪unfold⟫ f) (f v))) ∈ O'_F(F)(X)
 ⇐ { f_0 ∈ j f ∧ v_0 ∈ j v ⇒ f_0 v_0 ∈ j (f v). }
   ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_σ).
     ∀ f_0 ∈ j f, v_0 ∈ j (f v).
       (L(F) (⟦unfold⟧ f_0) v_0, J' (F (⟪unfold⟫ f) (f v))) ∈ O'_F(F)(X)
 ⇐ { Generalise. f ∈ dom(j) ∧ v ∈ dom(j) ⇒ f v ∈ dom(j). }
   ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_(F σ)).
     ∀ f_0 ∈ j f, v_0 ∈ j v.
       (L(F) (⟦unfold⟧ f_0) v_0, J' (F (⟪unfold⟫ f) v)) ∈ O'_F(F)(X)
 ⇐ { Generalise. }
   ∀ G ≤ F, f ∈ dom(j_(σ → F σ)), v ∈ dom(j_(G σ)).
     ∀ f_0 ∈ j f, v_0 ∈ j v.
       (L(G) (⟦unfold⟧ f_0) v_0, J' (G (⟪unfold⟫ f) v)) ∈ O'_F(G)(X)
 ⇐ { Induction over G. }
```

```
  ∀ G ≤ F.
    ( ∀ G' < G.
        ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_(G' σ)).
          ∀ $f_0$ ∈ j f, $v_0$ ∈ j v.
            (L(G') (⟦unfold⟧ $f_0$) $v_0$, J' (G' (⟪unfold⟫ f) v)) ∈ O'_F(G')(X)
    )
    ⇒ ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_(G σ)).
        ∀ $f_0$ ∈ j f, $v_0$ ∈ j v.
          (L(G) (⟦unfold⟧ $f_0$) $v_0$, J' (G (⟪unfold⟫ f) v)) ∈ O'_F(G)(X)
⇐ { Case analysis. }
```

- G = Id:

```
    ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_σ).
      ∀ $f_0$ ∈ j f, $v_0$ ∈ j v.
        (⟦unfold⟧ $f_0$ $v_0$, j' (⟪unfold⟫ f v)) ∈ X
  ⇔ { Definition of X. }
    ⊤
```

- G = K_τ:

```
    ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_τ).
      ∀ $f_0$ ∈ j f, $v_0$ ∈ j v.
        $v_0$ ∼ $j''$ v
  ⇔ { By definition of $j''$. }
    ⊤
```

- G = G_1 × G_2:

```
    ( ∀ G' < G.
        ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_(G' σ)).
          ∀ $f_0$ ∈ j f, $v_0$ ∈ j v.
            (L(G') (⟦unfold⟧ $f_0$) $v_0$, J' (G' (⟪unfold⟫ f) v)) ∈ O'_F(G')(X)
    )
    ⇒ ∀ f ∈ dom(j_(σ → F σ)), $v_1$ ∈ dom(j_($G_1$ σ)), $v_2$ ∈ dom(j_($G_2$ σ)).
        ∀ $f_0$ ∈ j f, $v_{01}$ ∈ j $v_1$, $v_{02}$ ∈ j $v_2$.
          ( (L($G_1$) (⟦unfold⟧ $f_0$) $v_{01}$, L($G_2$) (⟦unfold⟧ $f_0$) $v_{02}$)
          , (J' ($G_1$ (⟪unfold⟫ f) $v_1$), J' ($G_2$ (⟪unfold⟫ f) $v_2$))
          ) ∈ O'_F(G)(X)
  ⇔
    ( ∀ G' < G.
        ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_(G' σ)).
          ∀ $f_0$ ∈ j f, $v_0$ ∈ j v.
            (L(G') (⟦unfold⟧ $f_0$) $v_0$, J' (G' (⟪unfold⟫ f) v)) ∈ O'_F(G')(X)
    )
    ⇒ ∀ f ∈ dom(j_(σ → F σ)), $v_1$ ∈ dom(j_($G_1$ σ)), $v_2$ ∈ dom(j_($G_2$ σ)).
        ∀ $f_0$ ∈ j f, $v_{01}$ ∈ j $v_1$, $v_{02}$ ∈ j $v_2$.
          (L($G_1$) (⟦unfold⟧ $f_0$) $v_{01}$, J' ($G_1$ (⟪unfold⟫ f) $v_1$)) ∈ O'_F($G_1$)(X) ∧
          (L($G_2$) (⟦unfold⟧ $f_0$) $v_{02}$, J' ($G_2$ (⟪unfold⟫ f) $v_2$)) ∈ O'_F($G_2$)(X)
  ⇔
  ⊤

• G = G₁ + G₂:

  ( ∀ G' < G.
      ∀ f ∈ dom(j_(σ → F σ)), v ∈ dom(j_(G' σ)).
        ∀ f₀ ∈ j f, v₀ ∈ j v.
          (L(G') (⟦unfold⟧ f₀) v₀, J' (G' (⟪unfold⟫ f) v)) ∈ O'_F(G')(X)
  )
  ⇒ ∀ f ∈ dom(j_(σ → F σ)), f₀ ∈ j f.
      ∀ v₁ ∈ dom(j_(G₁ σ)), v₀₁ ∈ j v₁.
        (L(G₁) (⟦unfold⟧ f₀) v₀₁, J' (G₁ (⟪unfold⟫ f) v₁)) ∈ O'_F(G₁)(X)
      ∧
      ∀ v₂ ∈ dom(j_(G₂ σ)), v₀₂ ∈ j v₂.
        (L(G₂) (⟦unfold⟧ f₀) v₀₂, J' (G₂ (⟪unfold⟫ f) v₂)) ∈ O'_F(G₂)(X)
⇔
  ⊤
```

□

20 Strict language

The main theorem holds for a strict language as well

Simple definition of strict language: Same syntax, same semantics, except that

```
  ⟦t_1 t_2⟧_⊥ ρ = { (⟦t_1⟧_⊥ ρ) (⟦t_2⟧_⊥ ρ), ⟦t_2⟧_⊥ ρ ≠ ⊥,
                  { ⊥,                         otherwise.
```

Note: Strange strict language, includes coinductive types.

Translation:

```
        { seq t_2^* (t_1^* t_2^*), t = t_1 t_2,
  t^* = | λx. t_1^*,               t = λx. t_1,
        { t,                       otherwise.
```

It is straightforward to check that this translation is type-preserving.

We have ⟦t⟧_⊥ ρ = ⟦t^*⟧ ρ. Proof by induction over structure of t:

Application:

1. Assume that ⟦t_2⟧_⊥ ρ ≠ ⊥:

```
     ⟦t_1 t_2⟧_⊥ ρ
   = { ⟦t_2⟧_⊥ ρ ≠ ⊥. }
     (⟦t_1⟧_⊥ ρ) (⟦t_2⟧_⊥ ρ)
   = { Inductive hypothesis. }
     (⟦t_1^*⟧ ρ) (⟦t_2^*⟧ ρ)
   = { Inductive hypothesis, ⟦t_2^*⟧ ρ = ⟦t_2⟧_⊥ ρ ≠ ⊥. }
     ⟦seq⟧ (⟦t_2^*⟧ ρ) ((⟦t_1^*⟧ ρ) (⟦t_2^*⟧ ρ))
   =
     ⟦seq t_2^* (t_1^* t_2^*)⟧ ρ
   =
     ⟦(t_1 t_2)^*⟧ ρ
```

2. Assume that ⟦t_2⟧_⊥ ρ = ⊥:

```
     ⟦t_1 t_2⟧_⊥ ρ
   = { ⟦t_2⟧_⊥ ρ = ⊥. }
     ⊥
   = { Inductive hypothesis: ⟦t_2^*⟧ ρ = ⟦t_2⟧_⊥ ρ = ⊥. }
     ⟦seq⟧ (⟦t_2^*⟧ ρ) (⟦t_1^* t_2^*⟧ ρ)
   =
      ⟦seq t₂^* (t₁^* t₂^*)⟧ ρ
    =
      ⟦(t₁ t₂)^*⟧ ρ

  Abstraction:

    ⟦λx.t⟧_⊥ ρ
  =
    λv.⟦t⟧_⊥ ρ[x ↦ v]
  = { Inductive hypothesis. }
    λv.⟦t^*⟧ ρ[x ↦ v]
  =
    ⟦(λx.t)^*⟧ ρ

  Otherwise:

    ⟦t⟧_⊥ ρ
  =
    ⟦t⟧ ρ
  =
    ⟦t^*⟧ ρ
```

□

We also have ⟪t⟫ ρ = ⟪t^*⟫ ρ (easy induction over t).

Given the two results above we immediately get that the main result and its corollary hold when ⟦·⟧ is replaced by ⟦·⟧_⊥, with the following slightly modified precondition regarding uses of seq in the term in question:

• seq is not used in _the translation of_ the term at a type with ⊥ in its domain.

Proof:

  Note first that if t satisfies the precondition above, then both t and t^* satisfy the preconditions of the fundamental theorem and the main result regarding uses of seq at the wrong type.

  Now, given a term t and contexts ρ, ρ' satisfying all the preconditions of the main result, including the modified one, we get that ⟦t⟧_⊥ ρ ∈ dom(~) and hence:

```
    [⟦t⟧_⊥ ρ]
  = { Result above. }
    [⟦t^*⟧ ρ]
  = { Main result. }
```

```
    j (⟪t^*⟫ ρ')
  = { Result above. }
    j (⟪t⟫ ρ')
```

  Thus the main result holds for ⟦·⟧_⊥. The corollary follows in the same way as before.

□

Note that the translation above leads to an exponential blowup in term size. Probably this doesn't matter, since the translation won't be used in practice. However, we would get around the blowup by having

```
  (t₁ t₂)^* = seq t₂^* (t₁^* t₂)
```

instead of

```
  (t₁ t₂)^* = seq t₂^* (t₁^* t₂^*).
```

Is this possible? The answer is no, not if we want to have ⟦t⟧_⊥ ρ = ⟦t^*⟧ ρ.

Denote the new variant of the translation by ^⋆. Assume that ⟦t^⋆⟧ ρ ≠ ⊥. We get:

```
  ⟦((λx.x) t)^⋆⟧ ρ = ⟦(λx.x) t⟧_⊥ ρ
⇔
  ⟦seq⟧ (⟦t^⋆⟧ ρ) (⟦(λx.x)^⋆⟧ ρ (⟦t⟧ ρ)) = ⟦(λx.x) t⟧_⊥ ρ
⇔ { ⟦t^⋆⟧ ρ ≠ ⊥. }
  ⟦(λx.x)^⋆⟧ ρ (⟦t⟧ ρ) = ⟦(λx.x) t⟧_⊥ ρ
⇔
  ⟦t⟧ ρ = ⟦(λx.x) t⟧_⊥ ρ
⇔ { ⟦t⟧_⊥ ρ = ⊥ ⇒ ⟦(λx.x) t⟧_⊥ ρ = ⊥. }
  ⟦t⟧ ρ = ⟦t⟧_⊥ ρ
```

Hence, for an arbitrary term t we should have

```
  ⟦t^⋆⟧ ρ ≠ ⊥ ⇒ ⟦t⟧ ρ = ⟦t⟧_⊥ ρ.
```

It is easy to find a counterexample:

```
  t = λx.(λz.x) y, ρ = [y ↦ ⊥]

  ⟦t^⋆⟧ ρ = λv.⊥ ≠ ⊥

  ⟦t⟧ ρ = λv.v

  ⟦t⟧_⊥ ρ = λv.⊥
```

References

Nils Anders Danielsson, Jeremy Gibbons, John Hughes, and Patrik Jansson. Fast and loose reasoning is morally correct. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 206–217, 2006.